ABSTRACT

Title of dissertation: IMPROVING PROGRAM TESTING AND UNDERSTANDING VIA SYMBOLIC EXECUTION

Kin Keung Ma, Doctor of Philosophy, 2011

Dissertation directed by: Professor Jeffrey S. Foster
Professor Michael Hicks
Department of Computer Science

Symbolic execution is an automated technique for program testing that has recently become practical, thanks to advances in constraint solvers. Generally speaking, a symbolic executor interprets a program with symbolic inputs, systematically enumerating execution paths induced by the symbolic inputs and the program's control flow. In this dissertation, I discuss the architecture and implementation of Otter, a symbolic execution framework for C programs, and work that uses Otter to solve two program analysis problems.

Firstly, we use Otter to solve the line reachability problem—given a line target in a program, find inputs that drive the program to the line. We propose two new directed search strategies, one using a distance metric to guide symbolic execution towards the target, and another iteratively running symbolic execution from the start of the function containing the target, then jumping backward up the call chain to the start of the program. We compare variants of these strategies with several existing undirected strategies from the literature on a suite of 9 GNU Coreutils programs. We find that most directed strategies perform extremely well in many

cases, although they sometimes fail badly. However, by combining the distance metric with a random-path strategy, we obtain a strategy that performs best on average over our benchmarks. We also generalize the line reachability problem to multiple line targets, and evaluate our new strategies under a different experimental setup. The result shows that many directed strategies start off slightly slower than undirected strategies, but they catch up and perform the best in the long run.

Another use of Otter is to study how run-time configuration options affect the behavior of configurable software systems. We conjecture that, at certain levels of abstraction, software configuration spaces are much smaller than combinatorics might suggest. To evaluate our conjecture, we ran Otter on three configurable software systems with their concrete test suites, but making configuration options symbolic. Otter generated data of all execution paths of these systems, from which we discovered how the settings of configuration options affect line, basic block, edge, and condition coverage for our subjects under the test suites. Had we instead run the test suites under all configuration settings, it would have required many orders of magnitude more executions to generate the same data.

IMPROVING PROGRAM TESTING AND UNDERSTANDING VIA SYMBOLIC EXECUTION

by

Kin Keung Ma

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2011

Advisory Committee:
Professor Jeffrey S. Foster, Co-chair/Advisor
Professor Michael Hicks, Co-chair/Co-advisor
Professor Shuvra Bhattacharyya, Dean's Representative
Professor Adam Porter
Professor William Gasarch

To my parents

Acknowledgments

I would like to thank my advisors, Dr. Jeffrey Foster and Dr. Michael Hicks, for their teaching, guidance and funding. I would also like to thank my collaborators: Khoo Yit Phang, Elnatan Reisner, Jonathan Turpie, Dr. Adam Porter and Charles Song for their effort in making the studies in this dissertation successful. Lastly, I would like to express my deepest gratitude to my parents for their unending support and encourgement.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DSE	Directed symbolic execution
SDSE	Shortest-distance symbolic execution
SDSE-pr	Probabilistic SDSE
SDSE-intra	Intraprocedural SDSE
SDSE-rr	SDSE with round-robin distance-to-targets
RR(X,Y)	Round-robin strategies X and Y
B(X)	Batched strategy X
Ph(X,Y,r)	First use strategy X, then switch to Y based on factor r
RP	Random-path strategy
CCBSE(X)	Call-chain-backward symbolic execution, using X as the forward search strategy
Mix-CCBSE(X)	Mixing X with CCBSE(RP), with 75% of the time running X

Chapter 1

Introduction

Every year, billions of dollars are lost due to software system failures [51]. For example, in 2010, Toyota recalled more than 13 million vehicles worldwide due to a bug in its vehicles' software that gave faulty speed readings, costing Toyota an estimated 2-5 billion dollars [53]. As another example, the London Stock Exchanges IT system collapsed in 2007. The stock market was paused for 40 minutes due to the collapse, and as a result billions of pounds worth of share trades were lost [35].

More than a third of this cost could be avoided if better software testing was performed [51]. However, software testing comes with great cost. Typically, about half of the man-hours of a software project is dedicated to software testing. Considering that billions of dollars are spent on software development every year, more efficient and effective software testing processes are of great interest.

A huge body of work has studied designing *automated* solutions for program testing (see Chapter 5). *Symbolic execution* is one automated technique proposed back in the 1970s [28]. It remained an unrealized idea for decades, but recently it has become practical, thanks to advances in *constraint solvers* [21, 16] used to efficiently limit the search space. Generally speaking, a *symbolic executor* interprets a program with *symbolic inputs*, systematically enumerating execution *paths* induced by the symbolic inputs and the program's control flow. Unlike certain *black-box*

approaches (e.g.,[7]) that generate concrete tests randomly, symbolic executors only generate one path for each set of inputs that drive the program to the same path, and therefore they avoid repeated work. Also, by design, symbolic executors are *complete*—paths generated by a symbolic executor are always realizable. In other words, should a symbolic executor find a path that triggers a bug, the bug actually *exists*, and a bug-triggering input can be derived from the *path condition* (Chapter 2.2). Knowing how a bug manifests gives programmers great help for debugging it.

Programs often have an unbounded number of paths, so it is impossible to enumerate all of them. Much of the literature has focused on developing symbolic execution *search strategies* so that the "interesting" paths are explored first, where interest is defined by a goal, such as maximizing code coverage [11]. In Chapter 3, I will present work that uses symbolic execution to solve the *line reachability problem*—given some line(s) of code in a program, the goal is to find inputs that drive the program to those lines. This work has applications to program testing and analysis.

Another use of symbolic execution, although less common, is to fully enumerate *all* execution paths of a program given a *constrained* input (e.g., an input taken from a relatively small set of possible values). Therefore enumerating all paths is feasible—in the worst case there is one execution path per combination of input values. Furthermore, this exactly models *configurable software*, where *flags*, often booleans, are used to control the software's behavior. In Chapter 4, we shall see how to use symbolic execution to enhance understanding of configurable software.

1.1 Thesis

This work aims to develop a framework for symbolic execution and use it to assist program testing and understanding. Concisely, this dissertation shows that

> *Symbolic execution can be improved to (1) solve the line reachability problem effectively using directed search strategies, and (2) help understanding configurable software systems by incorporating symbolic execution with coverage analyses.*

In support for this thesis, we developed Otter, a symbolic execution framework for C programs. This dissertation describes the implementation of Otter and how it is used in two software analysis problems: solving the line reachability problem and understanding configurable software systems. For each problem, we discuss its motivation and applications, explain its complexity using examples, and present experimental results that show the effectiveness of our techniques. Finally, we suggest future work to improve symbolic execution's usefulness.

1.2 Contributions

The remainder of this section will sketch my contributions, which will be presented in the rest of this dissertation.

1.2.1 Otter, a Symbolic Execution Framework

Otter is a symbolic execution framework for C. Otter is written in OCaml, and employs the CIL (C Intermediate Language) infrastructure (version 1.3.7) to

transform a C program into a high-level representation [40]. Otter performs symbolic execution on the CIL representation, and uses STP as its constraint solver [21]. STP embeds the theory of bitvectors and arrays, which captures most expressions from the C language. In order to run Otter on programs that interact with the environment (e.g., I/O, environment variables), Otter is bundled with pre-configured system libraries. We import most of newlib [41] as the C library and we emulate part of the POSIX library ourselves.

Otter was also designed to easily adopt new search strategies and thus serves as a vehicle to compare them. We implemented a range of state-of-the-art strategies (random-path, KLEE [11] and SAGE [26]), and we also developed our own strategies, which are presented in Chapter 3.

1.2.2 Directed Symbolic Execution

We study the problem of automatically finding program executions that reach a particular target line. This problem arises in many debugging scenarios; for example, a developer may want to confirm that a bug reported by a static analysis tool on a particular line is a true positive, i.e., that can actually arise under realistic conditions. We propose two new classes of *directed* symbolic execution strategies that aim to solve this problem: *shortest-distance symbolic execution (SDSE)* uses a distance metric in an interprocedural control flow graph to guide symbolic execution toward a particular target; and *call-chain-backward symbolic execution (CCBSE)* iteratively runs forward symbolic execution, starting in the function containing the

target line, and then jumping backward up the call chain until it finds a feasible path from the start of the program. We also propose a hybrid strategy, Mix-CCBSE, which alternates CCBSE with another (forward) search strategy. We compare these three new strategies with several existing *undirected* strategies (KLEE, SAGE and random-path) from the literature on a suite of 9 GNU coreutils programs containing 10 bugs. We also generalize the line reachability problem to multiple line targets.

We find that SDSE strategies performs extremely well in many cases compared to undirected strategies, but they sometimes fail badly. CCBSEs and Mix-CCBSEs also perform quite well sometimes, but impose additional overhead that often makes them slower than SDSEs. Finally, we try to combine SDSE with random-path, and found this combination performed best on average over all our benchmarks, combining to good effect the features of its constituent components. We also find that directed strategies tend to perform very well on the multi-target line reachability problem. Often undirected strategies start off finding targets quickly, however directed strategies are able to increase coverage gradually, and get better coverage in the end.

To our best knowledge, this is also the first work to study

- Symbolic execution *in the middle of a program* (whereas prior symbolic execution work only starts from the beginning of a program, i.e., `main`, or from the beginning of a function with programmer-supplied *pre-conditions*);
- The randomness of symbolic execution strategies. By running the same test with different random seeds, we found that the performance of a strategy can

be highly variable.

1.2.3 Using Symbolic Execution to Understand Behavior in Configurable Software Systems

Many modern software systems are designed to be highly configurable, which increases flexibility but can make programs hard to test, analyze, and understand. We present an initial empirical study of how configuration options affect program behavior. Our goal is to show that, at certain levels of abstraction, configuration spaces are far smaller than the worst case, in which every configuration induces distinct behavior. We studied three configurable software systems: vsftpd, ngIRCd, and grep. We used symbolic execution to discover how the settings of run-time configuration options affect line, basic block, edge, and condition coverage for our subjects under a given test suite. Our results strongly suggest that for these subject programs, test suites, and configuration options, when abstracted in terms of the four coverage criteria above, configuration spaces are in fact much smaller than combinatorics would suggest and are effectively the composition of many small, self-contained groupings of options.

This is a collaborative work. Apart from developing Otter and its coverage tracking features, I was also in charge of the analysis on ngIRCd.

Chapter 2

Otter: A Framework for Symbolic Execution

In this chapter, I will present an overview of symbolic execution, followed by a detailed discussion of Otter's design and implementation.

2.1 Background

In the mid 1970's, King [28] introduced symbolic execution as an extension of normal execution that can be used to enhance testing. He described basic concepts of symbolic execution, such as path conditions, "forking" on unresolvable conditionals, and using ASSUME and ASSERT to specify program properties. King and his colleagues implemented his ideas as a prototype tool called EFFIGY, which applies symbolic execution to a small language. King showed that EFFIGY had promise, but only evaluated EFFIGY on a few small examples. Also, theorem provers were less powerful at that time, limiting EFFIGY's potential. For example, it did not deal with array reads or writes with symbolic indices.

Recent improvements to constraint solvers, both in efficiency and the ability to solve harder problems, have made symbolic execution a practical method for program analysis. In particular, researchers have developed powerful SMT solvers that support theories such as arithmetic, arrays, recursive datatypes and uninterpreted functions [21, 16]. As a result, one can express richer verification conditions

in symbolic execution. Recently, seveal symbolic executors [26, 24, 12, 11] that take advantage of these new capabilities were developed to address challenges in traditional software testing.

2.2 An Overview of Symbolic Execution

The term *symbolic execution* has different meanings in different settings. Informally, we understand symbolic execution as a way of interpreting programs that contain *symbolic values*. A symbolic value is defined by the symbol and the set of *concrete values* it can range over. For instance, we can define α to be a symbol that can range over any value from the set of all 32-bit integers (such a set can be viewed as the *type* of the symbol). To perform symbolic execution on C programs, we let variables store symbolic values (e.g., variable x stores symbol α rather than a concrete integer like 3). For the ease of comprehension, in the ongoing text we will use English letters for variables (e.g., x, y, z) and Greek letters for symbolic values (e.g., α, β, γ). Also, we will use the symbol $\mapsto$ to denote assignments of values to variables (e.g., $\mathsf{x} \mapsto 3$, $\mathsf{y} \mapsto \beta$).

To interpret a program with symbolic values, we have to extend the usual semantics of the program. For example, executing the statement $\mathsf{y} = \mathsf{x} + 3$ where $\mathsf{x} \mapsto \alpha$ should yield $\mathsf{y} \mapsto (\alpha + 3)$, a *symbolic expression*. The symbolic executor maintains the program state (simply *state* for short) throughout the execution. The state comprises two parts: *Var*, a mapping from variables to values which include symbolic expressions (e.g., after executing $\mathsf{y} = \mathsf{x} + 3$, *Var* becomes $\{\mathsf{x} \mapsto$

$\alpha, \mathsf{y} \mapsto (\alpha + 3)\}$) and a set of *constraints* on symbolic values. For example, we can constrain symbols by ranges (e.g., $\alpha > 0$, $1 \leq \beta < 10$), or constrain the relationship between symbols (e.g., $\alpha < \gamma$). Constraints on symbols can be provided as part of the program specification, or can be induced from the execution (we will see this shortly).

The symbolic executor runs a program in very much the same way as how an ordinary *interpreter* does. However, things start becoming different when it comes to *conditionals*, where the execution has to branch according to the state. In C, conditionals correspond to if-statements. An if-statement consists of a *condition*, which is an expression, a *true branch*, which is executed if the condition is evaluated to true, and a *false branch*, which is executed otherwise. If the condition is a symbolic expression, it could be that the condition may evaluate to either true or false, hence both branches could be feasible. To completely explore all possibilities, the symbolic executor must conceptually fork the execution to examine *both* branches. We will see an example of this branching shortly.

While we cannot avoid exploring both branches in general, we gain information when executing each branch, which may help us prune branches in the future. When symbolic execution follows the true branch, we know that the condition has to be true along the execution; similarly, if it follows the false branch, the condition has to be false. In other words, we impose constraints on the condition (a symbolic expression) in either branch. Figure 2.1a illustrates this idea. Suppose $\mathsf{x} \mapsto \alpha$ where α is a symbolic signed 32-bit integer. The program begins by testing if `x>0` in Line 2. Since α could be either positive or not, the execution forks and both branches are

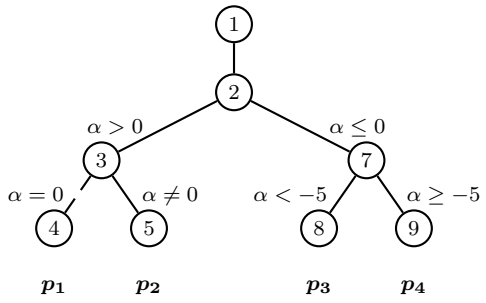
examined. On the true branch, it tests if `x==0`. Interestingly, we now know that `x` cannot be zero, since otherwise we would have followed the false branch. Therefore we are sure that the condition is evaluated to false, and thus the aborting failure in Line 4 is unreachable.

There are four *paths* explored while executing the code in Figure 2.1a symbolically. A path is defined to be a sequence of statements executed from the beginning of the program to the end. The set of all paths through a program forms a tree. For instance, the tree corresponding to the example code is shown in Figure 2.1b. Each node, labelled by the associated statement in the code, corresponds to a state in the symbolic execution. If a node has more than one child, the outgoing edges are labelled by the conditions that lead to the branching. The conjunction of all conditions seen from traversing from the root to a certain node is the *path condition* at that node. It describes the constraints that symbolic values must satisfy for execution to take that path. For example, the path condition at the node 9 associated with Line 9 is $(\alpha \leq 0) \wedge (\alpha \geq -5)$. Further symbolic execution along the path rooted at 9 must obey this path condition, e.g., any test of `x==c` where `c` is outside the range $[-5, 0]$ must yield false. Notice that path conditions of different paths are distinct, since otherwise there would exist some concrete input that drives the execution to two different paths, which is impossible. Furthermore, these path conditions *partition* the input space. For example, the four paths p_1, p_2, p_3 and p_4 correspond to input spaces $\{\}$, $\{\alpha > 0\}$, $\{\alpha < -5\}$ and $\{-5 \leq \alpha \leq 0\}$, respectively.

Constraint solvers are used to reason about symbolic expressions automatically. A constraint solver is a procedure that, given a set of constraints over variables,

```
1 int x=α; // symbolic
2 if(x>0){
3     if(x==0)
4         abort();
5     return 0;
6 }
7 else if(x<−5)
8     return 1;
9 // etc
```

(a)



(b)

Figure 2.1: An example and its path condition tree

finds an assignment of the variables that satisfy the constraints. Today, there are many types of constraint solvers available, and they vary in the problem domains that they are designed for. The choice of constraint solvers depends on the language and the nature of the program being executed. For example, to symbolically execute the program in Figure 2.1a and reason about the unreachability of Line 4, a Satisfiability Modulo Theories (SMT) solver with the theory of linear arithmetic is sufficient. To use an SMT solver, the problem to solve is transformed to an SMT *instance* that is passed to the solver. For exmaple, to determine if α could equal 0 under the path condition $\alpha > 0$, we construct the SMT instance $(\alpha > 0) \wedge (\alpha = 0)$ and let the SMT solver decide if this is satisfiable. It is not, as expected, and thus we can stop evaluating path p_1 during the execution (indicated by the dashed line).

To summarize, symbolic execution, in its simplest form described above, explores all possible paths in a program that a normal run can execute. No abstraction

on values is made, and therefore symbolic execution retains complete information of how values flow through the program. Moreover, in our example, while there are 2^{32} possible assignments to the symbolic value α, the symbolic execution explores only 3 paths (recall that p_1 is unrealizable). This shows an important property of symbolic execution—the complexity depends on the logic of the program, rather than the size of the input space, which tends to be astronomically big.

2.3 An Overview of Otter

Otter[1] is a symbolic execution for C [42]. Otter is written in OCaml, and employs the CIL (C Intermediate Language) infrastructure to transform a C program into a high-level representation [40]. CIL eliminates redundant C constructs and leaves a clean, distilled representation in the form of an OCaml data structure. Otter then performs symbolic execution on the CIL representation. Otter currently uses STP as the constraint solver [21]. STP is tailored for solving consraints related to bitvectors and arrays, which captures most expressions from the C language, and is thus very suitable for the purpose. STP has been used in other symbolic executors, such as EXE [12] and KLEE [11].

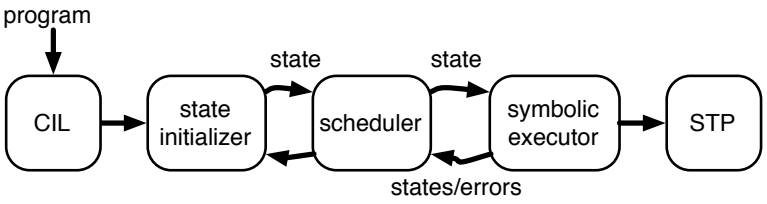
[1]DART [24] and EXE [12] are two well known symbolic executors. By coincidence, Dart and Exe are the names of two rivers in Devon, England. The others are the Otter, the Tamar, the Taw, the Teign, and the Torridge.

2.4 Architecture

Figure 2.2 diagrams the architecture of Otter and gives pseudocode for its main scheduling loop. Otter uses CIL to produce a control-flow graph from the input C program. Then it calls a *state initializer* to construct an initial symbolic execution *state*, which it stores in worklist, used by the scheduler. A state includes the stack, heap, program counter, and path taken to reach the current position. In traditional symbolic execution, which we call *forward* symbolic execution, the initial state begins execution at the start of main. The scheduler extracts a state from the worklist via pick and symbolically executes the next instruction by calling step. As Otter executes instructions, it may encounter conditionals whose guards depend on symbolic values. At these points, Otter queries STP to see if legal, concrete representations of the symbolic values could make either or both branches possible, and whether an error such as an assertion failure may occur. The symbolic executor will return these outcomes to the scheduler, and those that are *incomplete* (i.e., non-terminal) are added back to the worklist. The call to manage_targets is just for an extension of Otter, called CCBSE, which will be discussed in Section 3.1.2; the call to manage_targets is a no-op for forward symbolic execution.

2.5 Invoking Otter

Otter carries out symbolic execution in exactly the same way we described in Section 2.1. To visualize the process, we will demonstrate how Otter is used to symbolically execute the same example discussed in Section 2.1, but made into a



(a) Architecture diagram

```
1  scheduler()
2    while (worklist nonempty)
3      s0 = pick(worklist)
4      for s ∈ step(s0) do
5        if (s is incomplete)
6          put(worklist,s)
7        manage_targets(s)
```

(b) Scheduling loop

Figure 2.2: The architecture of the Otter symbolic execution engine.

complete C program, as shown in Figure 2.3a. The function __SYMBOLIC is an Otter built-in; it is used to fill a variable (passed with its address) with a purely symbolic value. Also, abort() is defined as __ASSERT(0), another Otter built-in that flags an error whenever the predicate does not hold (here the predicate is zero—it always fails).

Figure 2.3b shows Otter's verbose output. Lines are of the form

[*p*,*c*] *location*: *event*

which means "on path p whose path condition has c clauses, *event* happens at *location*", where *event* is either a statement at *location* (*file name* : *line number*) being executed, or a message (like "Ask STP..."). Paths are numbered from zero. When Otter forks a path into two at a conditional **if**(g), each path will be given a new

number. Also, each new path conjuncts its path condition with a new clause (g/¬g for the true/false branch). A clause g can also be unconditionally added into a path condition by calling __ASSUME(g). This is useful if we want to constrain a symbolic value at the beginning, e.g., __ASSUME(x>0) will make sure that the symbolic value stored in x is positive. The number c reflects the length of the path condition and hence the *depth* of the current state in the execution tree.

The execution starts at main as shown in Figure 2.3b. It checks if x > 0. Otter consults STP since x's value is symbolic, and STP indicates that the truth value of x>0 is *unknown*, meaning that both branches are feasible. Therefore Otter branches path 0 into paths 1 and 2 at example.c:5. In this demonstration, we use the *depth-first* strategy to explore paths: whenever a path forks into two, Otter always goes along the false branch, until it returns, and then it backtracks and goes along the true branch. Hence Otter follows path 1 next, executing example.c:10. It again consults STP and forks path 1 into paths 3 and 4, where path 3 terminates with a return of 2 (example.c:12) and path 4 with a return of 1 (example.c:11). Otter then backtracks to example.c:5 and explores path 2, the true branch. It consults STP for x==0 (example.c:6). This time STP can tell that the predicate is always false, since path 2's path condition carries the constraint x>0, and hence example.c:7 is skipped and path 2 returns 0 (example.c:8). Otter has now finished exploring all feasible paths (2, 3 and 4), and it reports that three paths ran into completion, and no paths ran into error.

Suppose we alter example.c:5 so that the comparison on line 5 is x>=0. The execution trace in Figure 2.3b deviates at Line 18, as shown in Figure 2.3c. Specifi-

```
1  #define abort() __ASSERT(0)
2  int main() {
3      int x;
4      __SYMBOLIC(&x);
5      if(x>0){
6          if(x==0)
7              abort();
8          return 0;
9      }
10     else if(x<−5)
11         return 1;
12     return 2;
13 }
```

(a) example.c from Figure 2.1a

```
1  [0,0] example.c:4 : Enter function main: int (void)
2  [0,0] example.c:5 : IF (x > 0)
3  [0,0] example.c:5 : Ask STP...
4  [0,0] example.c:5 : Unknown
5  [0,0] example.c:5 : Branching on x > 0 at example.c:5.
6  [0,0] example.c:5 : Path 1 is the false branch and path 2 is the true branch.
7  [1,1] example.c:10 : IF (x < −5)
8  [1,1] example.c:10 : Ask STP...
9  [1,1] example.c:10 : Unknown
10 [1,1] example.c:10 : Branching on x < −5 at example.c:10.
11 [1,1] example.c:10 : Path 3 is the false branch and path 4 is the true branch.
12 [3,2] example.c:12 : return (2);
13 [3,2] example.c:12 : Program execution finished.
14 [4,2] example.c:11 : return (1);
15 [4,2] example.c:11 : Program execution finished.
16 [2,1] example.c:6 : IF (x == 0)
17 [2,1] example.c:6 : Ask STP...
18 [2,1] example.c:6 : False
19 [2,1] example.c:8 : return (0);
20 [2,1] example.c:8 : Program execution finished.
21
22 3 paths ran to completion; 0 had errors.
```

(b) Otter's execution of (a)

```
16 [2,1] example.c:6 : IF (x == 0)
17 [2,1] example.c:6 : Ask STP...
18 [2,1] example.c:6 : Unknown
19 [2,1] example.c:6 : Branching on x == 0 at example.c:6.
20 [2,1] example.c:6 : Path 5 is the false branch and path 6 is the true branch.
21 [5,2] example.c:8 : return (0);
22 [5,2] example.c:8 : Program execution finished.
23 [6,2] example.c:7 : __ASSERT(0);
24 [6,2] example.c:7 : Error "'AssertionFailure: 0" occurs at example.c:7.
25 [6,2] example.c:7 : Abandoning path.
26
27 3 paths ran to completion; 1 had errors.
```

(c) Change in Otter's Output when Line 5 of (a) is changed to **if**(x>=0)

Figure 2.3: Invoking Otter

cally, x==0 becomes satisfiable and Otter forks. Path 6 hits the call to __ASSERT(0) at example.c:7, and Otter prints the error "AssertionFailure: 0" immediately, and reports that 1 path had error at the end.

2.6 Program States and Memory Model

A program state is a snapshot of the memory during the execution. Otter closely follows C's memory model, and therefore a state in Otter consists of the stack, heap and program counter, plus the path condition that led to it. The stack consists of stack frames, one for each active function call. A stack frame has a mapping from local variables to memory blocks (call this mapping VAR−BLOCK), and a pointer to an instruction in the caller function where the execution continues after this function returns. There is also one VAR−BLOCK for global variables. However, memory blocks associated with memory in the heap (i.e., created via calls to malloc) are not explicited stored in a VAR−BLOCK; their references implicitly exist as addresses stored in variables and in the heap itself (see Section 2.6.3).

Otter's program states are purely functional—Otter does not modify state in-place. Therefore, memory blocks do not directly "store" values. Instead, a program state has a mapping from memory blocks to the actual values they carry (called BLOCK−VAL). Having such design means that evaluating a variable is a two-step process: we first retrieve from a VAR−BLOCK the memory block associated with the variable, then retrieve from a BLOCK−VAL the value conceptually stored in the memory block. For example, with VAR−BLOCK=$\{\mathsf{x} \mapsto b_{\mathsf{x}}, \mathsf{y} \mapsto b_{\mathsf{y}}\}$

and BLOCK−VAL=$\{b_{\mathsf{x}} \mapsto 4, b_{\mathsf{y}} \mapsto \mathsf{ADDR}(b_{\mathsf{x}}, 0)\}$, x evaluates to 4 and y evaluates to &x ($\mathsf{ADDR}(b_{\mathsf{x}}, 0)$ denotes the address of b_{x} with zero offset; this will be discussed shortly). Note that VAR−BLOCK and BLOCK−VAL together function like *Var* discussed in Section 2.2. (The memory model in Section 2.2 was simplified to omit pointers.)

There are main advantages in having purely function program states. State creation is faster and uses less memory, thanks to persistent data structures, and backtracking does not require *undoing* state changes, thus program reasoning is easier.

2.6.1 Assumptions

Otter makes several assumptions to keep its design simple. Like most static analysis tools (e.g., [6]), it assumes that memory blocks are infinitely far apart, and so pointers cannot jump from one memory block to another. Also, Otter does not handle de facto standards not officially part of ANSI C, such as the ordering of fields in structs (although error-prone, we had seen programs relying on that).

2.6.2 Primitive Values

As in C, the byte is the basic unit of values in Otter. For instance, a symbolic integer comprises 4 *symbolic bytes*. This enables us to precisely model C memory operations. Consider the example in Figure 2.4a. Here, the call to __SYMBOLIC assigns n a sequence of 4 symbolic values $\alpha_0\alpha_1\alpha_2\alpha_3$, each α_i being a fresh symbolic

```
1 int n;
2 char *p = (char*)&n;
3 __SYMBOLIC(&n);
4 p[2] = 0;
```

(a)

```
1 int a,b;
2 __SYMBOLIC(&a);
3 __SYMBOLIC(&b);
4 __ASSUME(a<0);
5 __ASSUME(b>0);
6 __ASSERT(a<b);
7 __ASSERT((unsigned)a<(unsigned)b);
```

(b)

Figure 2.4: Examples

byte. Such a 4-byte integer can be viewed as a character array of length 4, so that each byte can be changed as shown in Line 4. After the execution, `n` will have value $\alpha_0\alpha_1 0\alpha_3$.

Having values represented by bytes also means that values are *untyped*. Figure 2.4b illustrates this idea. In this example, `a` and `b` are declared to be (signed) integers, and are set to hold a negative and positive symbolic integer, respectively. This is done by calling `__ASSUME` (Lines 4-5) to discard executions that have $\mathsf{a} \geq 0$ or $\mathsf{b} \leq 0$ (notice that, although the predicate involves variables, it is the symbolic values being held by variables that are constrained.) The program continues by calling `__ASSERT` (Lines 6-7) twice The first assertion checks `a<b` assuming they are signed integers. The second assertion checks almost the same thing, but assuming they are unsigned. Notice that the symbolic values stored in the variables are not constrained by the casts in the second assertion. However, the casts cause the less-than comparison to be performed differently, by not treating the operands as signed numbers. Therefore, the first assertion passes and the second one fails.

As an optimization, Otter represents constants as OCaml ints, until Otter needs to break them into byte arrays. For example, an expression of a constant 5 is stored as $\mathsf{CONST}(5)$, instead of a byte array $[05, 00, 00, 00]$ (Otter is little endian). The constant flows through the execution, until some expression reads/writes a part of it, in which case it is converted to a byte array.

2.6.3 Symbolic Expressions

Apart from primitive values, Otter also supports several different symbolic expressions summarized in Figure 2.5. They are:

Data pointers. A data pointer has the form $\mathsf{ADDR}(b, i)$, representing a pointer pointing to an offset i from the base address of the memory block b. The offset is an integer, which may be symbolic. When a pointer is dereferenced, the l-value is recovered as a portion of the memory block, determined by the offset and the type of the expression. Null pointers are represented by zeros (and they do not correspond to any memory blocks).

Function pointers. Function pointers are represented using a special symbolic expression $\mathsf{FUNPTR}(f)$ dedicated to function pointers, where an OCaml pointer to the CIL data structure of the function f is embedded and is retrieved when a function call through the pointer is made.

Operations on values. All unary and binary operators in C are supported. This is needed when at least one of the operands is symbolic. For example, an expression x+3 will be evaluated to $\mathsf{OP}(\mathsf{PLUS}, \alpha_{\mathsf{x}}, 3)$ where $\mathsf{x} \mapsto \alpha_{\mathsf{x}}$. Otherwise,

the expression will be evaluated as usual (see Section 2.7).

Array reads/writes. Whenever the array to be read/written is a symbolic expression (i.e., not a concrete byte array), or when the array index is symbolic, a symbolic expression will be created. For array reads, symbolic expressions are of the form $\mathsf{READ}(arr, i, s)$, denoting "read $[i, i+s)$ from array arr", where i is the index, which can be symbolic, and s is the size to be read, which must be concrete. All units are in bytes. Similarly, array writes are of the form $\mathsf{WRITE}(arr, i, s, v)$, where v is the (possibly symbolic) value written into $arr[i, i+s)$.

Note that array reads/writes should not to be confused with pointer dereferences:to create the symbolic expression of `a[i]`, Otter first finds the values stored in the *entire* array `a`. This step is basically a dereference which is carried out as usual. Once the value of `a` is computed (say α), Otter creates symbolic expressions $\mathsf{READ}(\alpha, i, s)$ or $\mathsf{WRITE}(\alpha, i, s, v)$.

Conditional Values. A conditional value c is of the form $\mathsf{COND}(g, e_1, e_2)$, where g is a *boolean* expression, and e_1 and e_2 are the expressions c evaluates to when g is true or false, respectively.

$$
\begin{array}{rll}
e \rightarrow & \vec{\alpha} & (\textit{Symbols}) \\
| & \mathsf{CONST}(c) & (\textit{Constant}) \\
| & \mathsf{ADDR}(b, i) & (\textit{Data pointer}) \\
| & \mathsf{FUNPTR}(f) & (\textit{Function pointer}) \\
| & \mathsf{OP}(op, \vec{e}) & (\textit{Operation}) \\
| & \mathsf{READ}(e, i, s) & (\textit{Array read}) \\
| & \mathsf{WRITE}(e, i, s, v) & (\textit{Array write}) \\
| & \mathsf{COND}(g, e_1, e_2) & (\textit{Conditional value})
\end{array}
$$

(a) Expression types

$$
\begin{array}{rll}
op \rightarrow & uop \mid binop & \\
uop \rightarrow & \mathsf{UMINUS} \mid \mathsf{BNOT} \mid \mathsf{LNOT} & (\textit{Negations}) \\
binop \rightarrow & \mathsf{PLUS} \mid \mathsf{SUB} \mid \mathsf{MULT} \mid \mathsf{DIV} \mid \mathsf{MOD} & (\textit{Arithmetics}) \\
| & \mathsf{LT} \mid \mathsf{GT} \mid \mathsf{LE} \mid \mathsf{GE} \mid \mathsf{EQ} \mid \mathsf{NE} & (\textit{Comparisons}) \\
| & \mathsf{BAND} \mid \mathsf{BXOR} \mid \mathsf{BOR} \mid \mathsf{LAND} \mid \mathsf{LOR} & (\textit{Bitwise/logical}) \\
| & \mathsf{LSL} \mid \mathsf{LSR} & (\textit{Shifts})
\end{array}
$$

(b) Operators

Figure 2.5: Symbolic expressions

2.7 Semantics

2.7.1 Evaluations of Expressions

A primitive operation of Otter is evaluating a (side-effect-free) C expression (such as `a+b` or `r[i]`) under a program state. The output is the value of the expression, either as a concrete value (e.g., 3) or as a symbolic expression (e.g., $\mathsf{OP}(\mathsf{PLUS}, \alpha, \beta)$, $\mathsf{READ}(\rho, \iota, 4)$). Otter recurses on the structure of a C expression when evaluating it. The C expression structure basically contains

Constants. E.g., 3, 'c'. They are evaluated to themselves.

l-values. E.g., x, a[i]. Otter first computes their *l-value*s. An l-value is a triple (b, i, w), where b is a memory block, i is the offset (possibly symbolic) and w is the (concrete) size of the l-value (in bytes). From the triple, the value is computed as READ(BLOCK−VAL(b),i,w).

Operations. E.g., a+b, x==y, which are evaluated to OP(PLUS,eval(a),eval(b)) and OP(EQ,eval(x),eval(y)), resp. (eval(x) denotes the evaluation of x.)

AddrOf. E.g., &x. Otter computes its l-value (b, i, w), and returns ADDR(b, i).

(Other C expressions, such as **sizeof** and *cast*s, are trivially handled and thus omitted.)

If an expression involves only concrete values, e.g., summation of two concrete integers, or reading a regular array with a concrete index (however the value being read can be symbolic), Otter simplifies it to a single concrete value (e.g., OP(PLUS,3,4) is simplified to 7).

Computing l-values can be very tricky, because it generally involves dereferences of addresses, but STP does not reason about dereferences (see Section 2.8), Therefore, Otter has to implement this logic. The common case is when the address to be dereferenced is of the regular form ADDR(b, i), in which case the l-value is ready recovered (the width w of the l-value comes from the type of the C expression). Otherwise, Otter issues a failure, indicating that it is unable to reason about dereferences of non-trivial symbolic expressions, except for the following:

1. For conditional values, Otter recursively dereferences all leaves in the conditional tree, and returns a *conditional* l-value (e.g., $\mathsf{COND}(g, (b_1, i_1, w_1), (b_2, i_1, w_2))$).

2. Using the above, an optimization is made to dereferencing a $\mathsf{READ}(arr, i, s)$ expression, by converting it into

$$\mathsf{COND}(i == 0, arr[0], \mathsf{COND}(i == 1, arr[1], ...arr[n]...))$$

i.e., a conditional tree that enumerates all the possible indices.

2.7.2 Executing Instructions

Instructions in C can be divided into control statements, assignments and function calls. Among control statements, conditional branches (i.e., if-else) are handled specially. Otter consults the constraint solver for the *ternary* value (true, false and *unknown*) of $g = 0$ where g is the guard of the conditional. If it is a known false/true, then the true/false branch will be followed. Otherwise, either branch is possible, and Otter generates two program states (i.e., it forks): one state with $g = 0$ added to the path condition, and the false branch will be followed, and another state with $g \neq 0$ added to the path condition, and the true branch will be followed. These states are put into the scheduler (Figure 2.2a), which decides the next state to be run.

Assignments involve the evaluation of the expression on the right-hand-side and the l-value of the left-hand-side, and is carried out via a change to $\mathsf{BLOCK{-}VAL}$. For example, given $\mathsf{VAR{-}BLOCK}=\{\mathsf{a} \mapsto b_{\mathsf{a}}, \mathsf{i} \mapsto b_{\mathsf{i}}\}$ and $\mathsf{BLOCK{-}VAL}=\{b_{\mathsf{a}} \mapsto \alpha, b_{\mathsf{i}} \mapsto \beta)\}$, and an assignment $\mathsf{a[i]} = 2$, Otter evaluates a to α and i to β, and changes

$\mathsf{BLOCK{-}VAL}$ to $\{b_{\mathsf{a}} \mapsto \mathsf{WRITE}(\alpha, \beta, 1, 2), b_{\mathsf{i}} \mapsto \beta)\}$.

Function calls are carried out by creating a *frame*, which is pushed onto the call stack. The frame consists of the program state with all formals carrying values from the evaluations of the arguments, and also a reference to the instruction in the callee to be run next, right after the function call is returned. Optionally it also specifies the l-value that is going to receive the returned value.

2.8 Interacting with the Solver

2.8.1 STP: an SMT Solver

STP is an SMT solver developed by Vijay Ganesh [21]. It is aimed at solving constraints generated by program analysis tools, theorem provers, automated bug finders, intelligent fuzzers and model checkers. The inputs to STP are formulas over the theory of bit-vectors and arrays (which captures most expressions from C), and the output of STP is a single bit of information that indicates whether the formula is satisfiable or not. If the input is satisfiable, then it can also generate a variable assignment to satisfy the input formula. STP is the backend constraint solver for many static analysis tools, including symbolic executors like EXE (co-designed with STP), KLEE and JPF-SE [5].

2.8.2 Converting Otter Expressions to STP Queries

Thanks to STP's support of bit-vectors and arrays, converting an Otter expression to an STP formula is mostly straightforward. For example, an expression

$\mathsf{OP}(\mathsf{PLUS}, \alpha, \beta)$, where α and β are symbolic 32-bit integers (4-byte symbolic arrays), is converted to an STP formula in the following steps:

1. Say $\alpha = \alpha_0\alpha_1\alpha_2\alpha_3$, where α_3 is the most significant byte (under little endian). Create a bitvector v_i of length 8 (size of a byte) for each α_i. Create $v_\alpha = v_\alpha^3 \mathbin{@} v_\alpha^2 \mathbin{@} v_\alpha^1 \mathbin{@} v_\alpha^0$ where $@$ denotes concatencation. v_α is 32 bits wide. Notice that the ordering of vectors is inverted due to STP's "big-endian" nature.

2. Similarly, create v_β for β.

3. Call the STP function BVPLUS(32,v_α,v_β); here 32 is the length of the bitvector operands.

Converting an expresion $\mathsf{READ}(arr, i, s)$ to an STP formula requires the use of STP arrays, which support array reads/writes with symbolic indices. Specifically, Otter first creates a new array arr with the same length as arr and each cell of length 8 (size of a byte). Then, Otter converts each byte of arr into an STP bit-vector which is assigned to a cell in arr. Lastly, it creates the STP formula by concatenating the cells $\mathsf{arr}[v_i] \mathbin{@} ... \mathbin{@} \mathsf{arr}[v_{i+s-1}]$ where v_j is the bit-vector of symbolic index j.

Certain Otter symbolic expressions ($\mathsf{ADDR}(b, i)$, function pointers, etc.) do not have STP equivalents. Otter is seldom required to convert these expressions to STP formulae (Otter handles nullity checks $\mathsf{ptr}{==}\mathsf{0}$ itself, thus does not consult STP). Should conversion be required, Otter assigns *concrete, random* and *unique* integer "addresses" to memory blocks and functions, and these numbers can be used in the conversions.

Otter uses STP to check if a guard expression is satisfiable assuming the path condition holds. Since the path condition is a conjunction of expressions ($e_1 \wedge e_2 \wedge ...$) collected along the path, converting it into an STP formula involves the same steps as discussed above. Finally, a query to STP is done by *asserting* the path condition and querying for the satisfiability of the guard expression.

As discussed earlier, STP does not model pointer dereferences, and therefore Otter handles dereferences (i.e., computing l-values) itself. More precisely, STP *does* handle pointer dereferences if we treat the whole memory as an array (i.e., all pointers are (symbolic) offsets to the base address of the whole memory). However, this does not scale well for most programs.

2.9 Error Checking

By design, Otter naturally flags errors when it fails to continue an execution path. In many cases, failures correspond to bugs, such as dereferencing an integer zero (i.e., a null pointer), or performing pointer subtraction with two pointers of different bases. Furthermore, Otter performs bounds checking—whether an index used to access an array is within the bounds of the array (i.e., it checks for buffer overflow). Otter does so by consulting STP for the bounding constraints (i.e., for $\mathsf{a[i]}$, the constraint $\mathsf{i} \geq 0 \wedge \mathsf{i} < |\mathsf{a}|$ where $|a|$ is the length of array a).

Moreover, should there be a *partial* error, Otter identifies the condition that causes the error and flags it, and lets the execution continue under the condition of where no error occurs. For example, suppose arr is an array of length 5 and i is an

index carrying an unconstrained symbolic value α_i. Then, an access arr[i] will cause the current execution path to split into two:

1. An erroneous path with condition $(\alpha_i < 0 \vee \alpha_i \geq 5)$; this path is abandoned immediately;

2. Another path with condition $(0 \leq \alpha_i < 5)$, which is added into the path condition, and the execution continues.

For the second path, it is crucial to add the condition into the path condition, so that whenever arr[i] appears again in the future, Otter knows that i at that moment does not cause a buffer overflow.

2.10 Optimizations

Otter implements a range of optimizations. Most of them, as suggested by other researchers in the literature, aim at using the constraint solver more intelligently, since it demands a lot of computation resources. This is done by avoiding calling the constraint solver, or by simplifying queries before passing to the solver.

One optimization is *relevant path condition extraction* suggested by KLEE [11]. We observed that most of the time only a small portion of the path condition is relevant to the expression to be evaluated. Recall that the path condition is the conjunction of a list of assumed conditions along a path. To find the relevant path condition, we construct a graph with conditions and the expression e to be evaluated as nodes, and add an edge between any two nodes that involve some common symbolic values. Then, the transitive closure rooted at e contains all conditions in the

relevant path condition. By only asserting the relevant path condition when determining the feasibility of a guard, we significantly lighten the load on the constraint solver.

Another optimization that works in conjunction with relevant path condition extraction is *query caching*. As its name suggests, we cache the results as true, false, or unknown of queries of the form (path condition, guard expression). This drastically improves the performance, as expressions are often evaluated more than once under the same relevant path condition.

Another optimization technique, which is commonly employed by other symbolic executors, originates from Lisp's *hash cons(tructor)*, where a structure is constructed only once. In Otter, structures are created for symbolic expressions. Without hash consing, we would construct an expression such as $\alpha+\beta$, and later construct the same expression but in a fresh structure, e.g., when the C expression `a+b` is evaluated repeatedly. Such duplication increases memory consumption and computation complexity. With hash consing, however, structures are put into a hash table, and later when the same structure is needed, instead of constructing a fresh copy of it, the old one in the hash table will be used. Hash consing improves memory usage (by not duplicating objects of the same structure), and structural equality essentially becomes physical (i.e., pointer) equality, which can be checked more quickly. The trade-off, however, is the overhead of calling a hash function whenever a structure is created. Nevertheless, this optimization often leads to better performance [11], and we find this to be the case in our experience.

2.11 Search Strategies

Search strategies refer to the way a scheduler (Figure 2.2a) assigns priorities to program states in order to achieve a certain goal (e.g., increase code coverage given a fixed amount of time). Symbolic execution can be thought as an exploration of a program's execution tree (e.g., Figure 2.1b), where nodes correspond to program states, and a node branches if the associated program state is forked into more than one state after execution. A search strategy determines in which order such execution tree is explored.

Unless symbolic execution is used for program verification, i.e., it traverses the entire execution tree (e.g., JPF-SE [5]), the search strategy determines how fast a goal is reached. Since almost all programs have unbounded execution trees in practice, search strategies play an important role in making symbolic execution practical.

Existing symbolic executors have used a variety of search strategies, each having its own rationale. For example, KLEE's search strategy is a mixture of (1) random exploration according to path length in the execution tree, and (2) a distance heuristic biasing towards program states that quickly lead to uncovered code according to the control flow graph. Thanks to Otter's search strategies framework, several state-of-the-art search strategies, such as KLEE and SAGE are implemented easily in Otter. A detailed discussion of these strategies is presented in Section 2.13.

Under Otter's search strategies framework, a strategy supports two operations: to put a program state into the scheduler, and to get the next state to be executed.

To make strategies composable, e.g., round-robin, where the ith strategy out of n strategies is used in $(kn+i)$-th iteration, each strategy must also support the remove operation.

Batching. We observe that for a search strategy to be effective, it must be highly efficient, because it is queried in every iteration (Figure 3.1). A strategy that has to look at all states in each iteration much too inefficient in practice. One way to cope with potential inefficiency is called *batching*, previously employed by KLEE. With batching, Otter continuously follows a path without considering other paths (therefore does not consult the search strategy), until the path forks, or the path is followed for a certain number of steps. This decreases the number of times the search strategy is consulted and therefore it greatly improves performance (in terms of the time spent on the search strategy). However, batching *alters* a strategy, and it is possible that, with batching, Otter spends too much time on paths that are not truly interesting, decreasing the strategy's *effectiveness*. Hence Otter makes batching an option to the user.

2.12 Interacting with the Environment

Programs interact with the system enviroment in a variety of ways. Examples are getting input from the console/files, reading environment variables, and outputting to the console/files. The code that facilitates these interactions is usually provided by the system as a library, such as libc and the POSIX libraries. In order to symbolically execute a realistic program, a model of the system library (at least,

a portion of the library used by the program) must be provided[2]. Thus, to make Otter convenient to use, we bundle Otter with a default model of system library.

We could either implement our own libc/POSIX, or import an existing implementation from elsewhere (such as glibc [2]). The advantage of implementing our own is we have full control of the complexity of the implementation. In particular, optimizations commonly applied in existing implemenations can actually hurt the performance when executed symbolically, and optimizations are often complex (e.g., different code optimized for different hardwares), making them very hard to port to Otter. On the other hand, reimplementing our own libraries is a time-consuming and error-prone task (considering that existing implemntations take many human-hours to develop and test).

Our solution is to do both. For libc we chose to import an existing implementation called newlib [41]. newlib is a C library intended for use on embedded systems. As a result, it is highly portable, and requires very few modifications to work well with Otter. For POSIX, it is much harder to find a working implementation, since POSIX includes many system calls that have to be defined in Otter. Therefore, we implemented a partial model of POSIX system calls. This includes an *in-memory* file system (where a file's content is stored in a **char** array), and functions that emulate system calls, such as network I/O, select (synchronous I/O multiplexing), and a subset of functions defined in unistd.h.

Notice that most of the library code is written purely in C, and therefore Otter

[2] Another symbolic execution paradigm, called *concolic testing*, models the environment differently. This will be discussed in Section 2.13.

executes it in the same way as any other source code, e.g., strcpy from newlib copies characters using a for-loop. A few functions, such as those defined in setjmp.h, require special supports from Otter (e.g., to implement setjmp, Otter has to remember the calling environment, which is later used by longjmp to restore the environment).

2.13 Related Work

In this section, I will introduce several symbolic executors from the literature, and compare them to Otter.

2.13.1 EXE and KLEE

EXE [12] was a symbolic executor developed in 2006 at Stanford Univesity. EXE instruments C programs by adding code that maintains symbolic constraints along execution paths, consults a constraint solver (STP) when a conditional is hit, and calls fork to branch the execution if the conditional is unresolvable. The instrumented program is then compiled and run *natively*.

KLEE [11], the successor to EXE, performs symbolic execution in a similar manner. However, instead of instrumenting the program and running it natively, KLEE *interprets* it. The main advantage of this over calling fork is that the latter requires duplication of memory, which is expensive in both time and space (although fork does copy-on-write, it is likely that any branch will modify memory, which triggers the copy). KLEE avoids this by modeling memory as a persistent map so that portions of the heap can be efficiently shared among multiple executions.

EXE and KLEE are able to find inputs that crash various programs, including a DHCP server, a regular expression library, several Linux file systems, and the GNU Coreutils suite [15].

Otter is similar to KLEE in that it also interprets programs, and it uses STP as the constraint solver. Several major differences between Otter and KLEE are

Environment modeling. KLEE uses uClibc [54] rather than newlib as the standard C library. Furthermore, KLEE also comes with an in-memory symbolic file system, but it only supports a flat, single directory structure (whereas Otter's file system supports hierarchical directory structures). It is also closely tied to the file system: whenever a program maniputes a symbolic file (e.g., opens a file given its symbolic name), KLEE creates *real* files in its sandbox in the actual file system. One consequence of this design is that the model is less portable—currently KLEE can only be run on Linux if POSIX support is required, whereas Otter does not have this limitation. Nevertheless, KLEE has special supoprt for *concrete* files: any file system calls with concrete filenames go directly to the real file system. This leads to much faster execution on file system calls with common files (e.g., /etc/fstab).

Strategies. KLEE uses a strategy that combines two strategies, called *random path selection* and *coverage-optimized search*, in a round-robin fashion.

- *Random path selection(RP)* [10] is a probabilistic version of breadth-first search. RP randomly chooses from the worklist states, weighing a state with a path of length n by 2^{-n}. Thus, this approach favors shorter paths,

but treats all paths of the same length equally.

- *Coverage-optimized search* computes the distance between the end of each state's path and the closest uncovered node in the interprocedural control-flow graph, and then randomly chooses from the set of states weighed inversely by distance. (To our knowledge, this algorithm has not been described in detail in the literature; we studied it by examining KLEE's source code [29].)

On the other hand, Otter favors flexible strategy deployment, while it is unclear if KLEE does. We implemented KLEE's strategy in Otter, and we compared it (as well as random path alone) against Otter's own strategies (Chapter 3).

Compilation framework. KLEE uses LLVM [34] to compile a C program into *bytecode* that is close to an assembly program, while Otter uses CIL to transform a C program to an intermediate representation that is a dialect of C.

Extensions. Otter has several extensions. In particular, one extension, called *call-chain-backward* symbolic execution (CCBSE, discussed in Chapter 3), requires starting symbolic execution *in the middle of a program*, and therefore requires support for conditional pointers and *lazy initialization*. To support these features, Otter needs a more sophisticated memory model and execution semantics. KLEE does not support starting symbolic in the middle of a program, and therefore we believe that KLEE does not support these features.

2.13.2 Concolic Testing

DART [24] combines random testing and symbolic execution to yield *concolic* testing (**conc**rete + symb**olic**). DART associates each symbolic input with a concrete value, and the program is executed *natively* with these values. At the same time, DART collects a list of symbolic constraints over the symbolic inputs, one at each branch point (i.e., conditional) along the concrete execution path. After the execution finishes, DART picks a branch point and negates the symbolic constraint. The new list of symbolic constraints is then put into a constraint solver, which generates a new input that will direct the program to another path with the same prefix as the previous one, but branching differently at the chosen branch point. This process is repeated until all branch points on all execution paths have been chosen, or it reaches maximum number of allowed paths.

DART uses lp_solve, which is a linear arithmetic constraint solver that does not solve constraints with pointers. If such constraints are present, DART simply reverts to ordinary random testing. CUTE [52] extends DART by improving its handling of pointer (in)equalities of the form $x = y$, $x \neq y$, $x =$ NULL and $x \neq$ NULL, and is able to discover errors such as memory leaks, segmentation faults and infinite loops. Hybrid concolic testing [37] further optimizes concolic testing by generating random inputs in the first phase to bring the symbolic execution to a certain state, and then, at that point, running concolic testing. The insight is if path explosion occurs at the very beginning of symbolic execution, ordinary concolic testing will "get stuck" in a small fraction of branches, those that can be reached using "short"

executions from the initial state of the program. Thus, hybrid concolic testing can improve the quality of branch coverage. Along the same lines, another paper [23] proposes fuzzing domain specific applications. By cooperating with a context-free constraint solver (which solves for satisfying assignments in the language accepted by some grammar), it dramatically improve code coverage when testing some Internet Explorer 7 interpreter modules.

SAGE [26], developed at Microsoft Research, also performs concolic testing. It has two major improvements over prior concolic testers:

Coverage-guided strategy. SAGE uses a coverage-guided *generational* search to explore states in the execution tree. Initially, at the *zeroth generation*, SAGE runs with the initial state; whenever the symbolic execution forks, SAGE chooses a branch at random to continue the execution, and stores the remaining branches into the worklist as the *first generation children*. After the zeroth generation finishes, SAGE runs each of the first generation children to completion, in the same manner as the zeroth generation, but separately grouping the grandchildren by their first generation parent. After exploring the first generation, SAGE explores subsequent generations (children of the first generation, grandchildren of the first generation, etc.) in a more intermixed fashion, using a block coverage heuristic to determine which generations to explore first.

Constraint solver. SAGE uses Z3 [16], a high-performance SMT solver also developed at Microsoft Research.

With these improvements, SAGE is reported to be very effective, and used daily by

Microsoft [26]. SAGE is not available in public.

2.13.2.1 Comparing Concolic Testing and Pure Symbolic Execution

The concolic testing literature refers to KLEE (and would refer to Otter) as *static* symbolic execution, while concolic testing itself is categorized as *dynamic* symbolic execution [26]. The author, however, does not agree with this classification: although KLEE and Otter do not natively execute programs, they both *interpret* programs without the conservative abstractions found in most static analysis tools. Furthermore, in theory, KLEE and Otter have the same *exploring power* as concolic testing, i.e., they explore the same program execution tree (though with different search order). In the following, I shall categorize EXE, KLEE and Otter as *pure symbolic execution*.

Concolic testing has several advantages over pure symbolic execution. First, concolic testing does native program execution, which is much faster than program interpretation, and it avoids the work of engineering an interpreter. It also handles environment modeling more naturally, since the environment (file system, network, etc.) is concrete and native. Second, a concolic tester consults its constraint solver only once per execution *path* to generate a new input for the next iteration, while a pure symbolic executor invokes its solver at every *conditional* that requires resolution. Considering that constraint solving is a major performance bottleneck, concolic testing's approach can be a great advantage. That said, both KLEE and Otter employ query caching (Section 2.10) to leverage the cost of frequent solver

queries. Furthermore, we expect more complex queries are generated from completed execution paths (required by concolic testing) than queries generated *in the middle* of executions (required by eurely symbolic executors).

On the other hand, pure symbolic execution has certain advantages over concolic testing. Most importantly, search strategies can be more flexible under pure symbolic execution. For example, a concolic tester can waste time exploring uninteresting paths, because it always executes the program into completion. However, a pure symbolic executor can *pause* a path and explore another one, and later come back to the first one again.

Moreover, since concolic testers run programs natively, they affect the external world, which makes them tricky to implement correctly and safely (e.g., consider two paths, one of which reads a file and another writes to the same file). For the same reason, it is hard for concolic testers to find errors related to edge cases of a system (e.g., out of memory, out of disk space, network failure, etc.) that do not normally happen.

Lastly, variants of symbolic execution, in particular CCBSE, are harder to implement using concolic testing.

2.13.3 Symbolic Execution for Exhausive Search

JPF-SE [5, 45] is a symbolic executor for Java programs. It is an extension of Java PathFinder (JPF), an explicit state model checking tool [3]. JPF-SE also performs pure symbolic execution. However, unlike KLEE and Otter, JPF-SE was

designed to always *exhaustively* enumerate all execution paths, and is therefore different from KLEE and Otter by the following:

- To cope with unbounded program executions due to loops and recursive calls, JPF-SE explores paths up to a certain *depth* provided by the user. KLEE and Otter, however, stops executing when reaching the time limit.

- Search strategies do not matter for JPF-SE, since it explores *all* paths. Currently, JPF-SE traverses the execution tree in a *depth-first* manner.

- JPF-SE is described as best for *unit* testing (instead of whole-program testing), or for programs that are small or contain no loops or recursive calls.

Chapter 3

Directed Symbolic Execution

In this chapter, we study the *line reachability problem*: given a target line in the program, can we find a realizable path to that line? Since program lines can be guarded by conditionals that check arbitrary properties of the current program state, this problem is equivalent to the very general problem of finding a path that causes the program to enter a particular state [25]. The line reachability problem arises naturally in several scenarios. For example, users of static-analysis-based bug finding tools need to *triage* the tools' bug reports—determine whether they correspond to actual errors—and this task often involves checking line reachability. As another example, a developer might receive a report of an error at some particular line (e.g., an assertion failure that resulted in an error message at that line) without an accompanying test case. To reproduce the error, the developer needs to find a realizable path to the appropriate line. Finally, when trying to understand an unfamiliar code base, it is often useful to discover under what circumstances particular lines of code are executed.

Symbolic execution is an attractive approach to solving line reachability: by design, symbolic executors are *complete*, meaning any path they find is realizable. However, symbolic executors cannot explore all program paths, and hence must make heuristic choices to prioritize path exploration. In this dissertation, we focus

on finding paths that reach certain lines in particular, whereas most prior work has focused on finding paths to increase code coverage [24, 12, 11, 37, 10, 55]. We are aware of one previously proposed approach, *execution synthesis* (ESD) [57], for using symbolic execution to solve the line reachability problem; we compare ESD to our work in Section 3.3.

We propose two new *directed* symbolic execution (DSE) search strategies for line reachability. First, we propose *shortest-distance symbolic execution* (SDSE), which prioritizes the path with the shortest distance to the target line as computed over an interprocedural control-flow graph (ICFG). Variations of this heuristic can be found in existing symbolic executors—in fact, SDSE is inspired by the heuristic used in the coverage-based search strategy from KLEE [11]—but, as far as we are aware, the strategy we present has not been specifically described nor has it been applied to directed symbolic execution. In Section 3.1.1 we describe how distance can be computed context-sensitively using *PN* grammars [50, 20, 48]. We will also discuss several variants of SDSE.

Second, we propose *call-chain-backward symbolic execution (CCBSE)*, which starts at the target line and works backward until it finds a realizable path from the start of the program, using standard forward (interprocedural) symbolic execution as a subroutine. More specifically, suppose the target line ℓ is inside function f. CCBSE begins forward symbolic execution from the start of f, yielding a set of partial interprocedural paths $\overline{p}_f$ that start at f, possibly call other functions, and lead to ℓ; in a sense, these partial paths summarize selected behavior of f. Next, CCBSE runs forward symbolic execution from the start of each function g that calls f, searching

for paths that end at calls to f. For each such path p, it attempts to continue down paths p' in $\overline{p}_f$ until reaching ℓ, adding all feasible extended paths $p + p'$ to $\overline{p}_g$. The process continues backward up the call chain until CCBSE finds a path from the start of the program to ℓ. Notice that by using partial paths to summarize function behavior, CCBSE can reuse the machinery of symbolic execution to concatenate paths together. This is technically far simpler than more standard approaches that use some formal language to explicitly summarize function behavior in terms of parameters, return value, global variables, and the heap (including pointers and aliasing).

The key insight motivating CCBSE is that the closer forward symbolic execution starts relative to the target line, the better the chance it finds paths to that line. If we are searching for a line that is only reachable on a few paths along which many branches are possible, then combinatorially there is a very small chance that a standard symbolic executor will make the right choices and find that line. By starting closer to the line we are searching for, CCBSE explores shorter paths with fewer branches, and so is more likely to reach that line.

CCBSE imposes some additional overhead, and so it does not always perform as well as a forward execution strategy. Thus, we also introduce *mixed-strategy CCBSE (Mix-CCBSE)*, which combines CCBSE with another forward search. In Mix-CCBSE, we alternate CCBSE with some forward search strategy S. If S encounters a path p that was constructed in CCBSE, we try to follow p to see if we can reach the target line, in addition to continuing S normally. In this way, Mix-CCBSE can perform better than CCBSE and S run separately—compared to

CCBSE, it can jump over many function calls from the program start to reach the paths being constructed; and compared to S, it can short-circuit the search once it encounters a path built up by CCBSE.

We implemented SDSE, CCBSE, and Mix-CCBSE in Otter. We also extended Otter with two popular forward search strategies, OtterKLEE and OtterSAGE, from KLEE [11] and SAGE [26], respectively. And, for a baseline, we implemented a random path search (RP) that flips a coin at each branch. We evaluated the effectiveness of our directed search strategies on the line reachability problem, comparing against the existing search strategies. We ran each strategy on 10 benchmarks from the GNU Coreutils programs [15], looking in each program for one line that contains a previously identified fault. We also compared the strategies on synthetic examples intended to illustrate the strengths of SDSE and CCBSE.

We found that SDSE and its variants perform extremely well on some programs, but it can fail completely under certain program patterns. CCBSE has performance comparable to standard search strategies but is often somewhat slower due to the overhead of checking path feasibility. Mix-CCBSE performs well on some of the benchmarks, particularly when using OtterKLEE as its forward search strategy, but it also fails in some cases. Lastly, we found that mixing SDSE with random-path gives the best strategy in terms of total time used across all benchmarks.

We also generalize our solutions to the line reachability problem to consider multiple line targets. More specifically, the *multi-target line reachability problem* is, given a time limit, find as many of a given set of line targets as possible. We implemented variants of SDSE, and compared them against other strategies using

the same set of benchmark programs from Coreutils, with line targets defined as lines not covered by Coreutils' test suite. We observe good performances from Mix-CCBSEs and mixing SDSEs with random-path. These results suggest that directed symbolic execution is a practical and effective approach to solving the line reachability problems.

3.1 Directed Strategies and Their Implementation

In this section we present SDSE, CCBSE, and Mix-CCBSE. We will explain them in terms of their implementation in Otter.

3.1.1 Shortest-Distance Symbolic Execution

The basic idea of SDSE is to prioritize program branches that correspond to the shortest path-to-target in the interprocedural CFG. To illustrate how SDSE works, consider the code in Figure 3.1, which performs command-line argument processing followed by some program logic, a pattern common to many programs. This program first enters a loop that iterates up to `argc` times, processing the i^{th} command-line argument in `argv` during iteration `i`. If the argument is `'b'`, the program sets `b[n]` to `1` and increments `n` (line 8); otherwise, the program calls `foo`. A potential buffer overflow could occur at line 8 when more than four arguments are `'b'`; we add an assertion on line 7 to identify when this overflow would occur. After the arguments are processed, the program enters a loop that reads and processes character inputs (lines 12 onward).

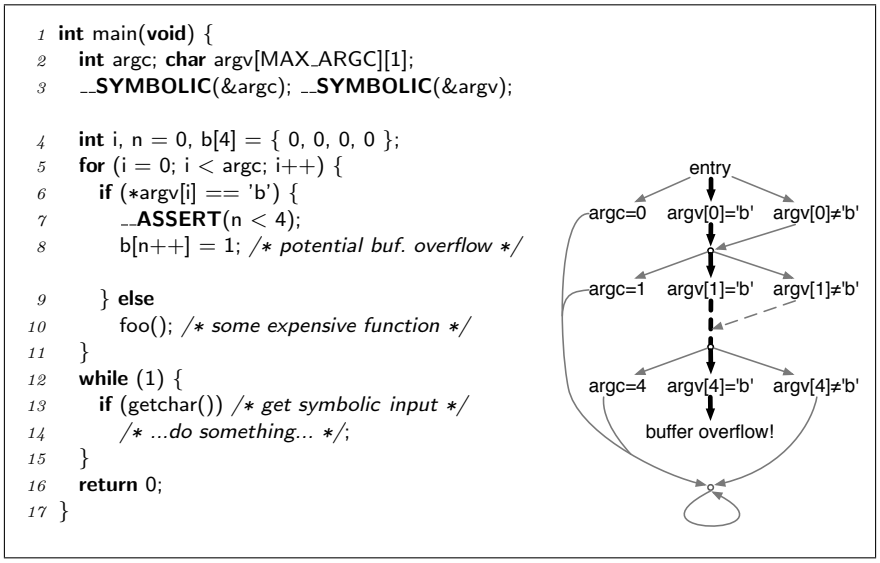


Figure 3.1: Example illustrating SDSE's potential benefit.

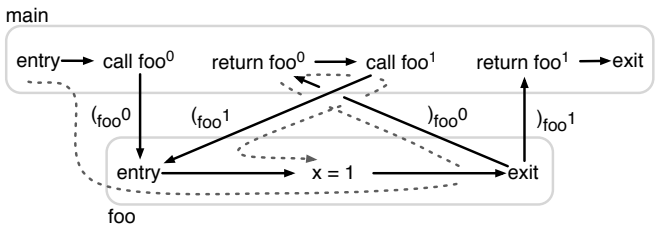
Suppose we would like to reason about a possible failure of the assertion. Then we can run this program with symbolic inputs, which we identify with the calls on line 3 to the built-in function __SYMBOLIC. The right half of the figure illustrates the possible program paths the symbolic executor can explore on the first five iterations of the argument-processing loop. Notice that for five loop iterations there is only one path that reaches the failing assertion out of $\sum_{n=0}^{4} 3 \times 2^n = 93$ total paths. Moreover, the assertion is not reachable once exploration has advanced past the argument-processing loop.

In this example, random-path (RP) would have only a small chance of finding the overflow, spending most of its time exploring paths shorter than the one that leads to the buffer overflow. OtterKLEE and OtterSAGE would focus on increasing coverage to all lines, wasting significant time exploring paths through the loop at the end of the program, which does not influence this buffer overflow.

In contrast, SDSE works very well in this example, with line 7 set as the target. Consider the first iteration of the loop. Otter will branch upon reaching the loop guard, and will choose to execute the first instruction of the loop, which is two lines away from the assertion, rather than the first instruction after the loop, which can no longer reach the assertion. Next, on line 6, the symbolic executor takes the true branch, since that reaches the assertion immediately. Then, determining that the assertion is true, it will run the next line, since it is only three lines away from the assertion and hence closer than paths that go through foo (which were deferred by the choice to go to the assertion). Then Otter will return to the loop entry, repeating the same process for subsequent iterations. As a result, SDSE explores the central path shown in bold in the figure, and thereby quickly finds the assertion failure.

Implementation. SDSE is implemented as a pick function from Figure 2.2. As mentioned, SDSE chooses the state on the worklist with the shortest *distance to target*. Within a function, the distance is just the number of edges between statements in the control flow graph (CFG). To measure distances across function calls, we count edges in an interprocedural control-flow graph (ICFG) [33], in which function call sites are split into *call nodes* and *return nodes*, with *call edges* connecting call nodes to function entries and *return edges* connecting function exits to return nodes. For each call site i, we label call and return edges by $(_i$ and $)_i$, respectively. Figure 3.2a shows an example ICFG for a program in which main calls foo twice; here call i to foo is labeled foo^i.

We define the distance-to-target metric to be the length of the shortest path



(a) Example PN-path in an interprocedural CFG.

$$
\begin{aligned}
PN &\rightarrow P\ N \\
P &\rightarrow S\ P \\
&\mid\ )_i\ P \\
&\mid\ \epsilon
\end{aligned}
\qquad
\begin{aligned}
N &\rightarrow S\ N \\
&\mid\ (_i\ N \\
&\mid\ \epsilon \\
S &\rightarrow (_i\ S\ )_i \\
&\mid\ S\ S \\
&\mid\ \epsilon
\end{aligned}
$$

(b) Grammar of PN paths.

Figure 3.2: SDSE distance computation.

in the ICFG from an instruction to the target, such that the path contains no mismatched calls and returns. Formally, we can define such paths as those whose sequence of edge labels form a string produced from the PN nonterminal in the grammar shown in Figure 3.2b. In this grammar, developed by Reps [50] and later named by Fähndrich et al [20, 48], S-paths correspond to those that exactly match calls and returns; N-paths correspond to entering functions only; and P-paths correspond to exiting functions only. For example, the dotted path in Figure 3.2a is a PN-path: it traverses the matching $(_{foo^0}$ and $)_{foo^0}$ edges, and then traverses $(_{foo^1}$ to the target. Notice that we avoid conflating edges of different call sites by matching $(_i$ and $)_i$ edges, and thus we can statically compute a context-sensitive distance-to-target metric.

PN-reachability was previously used for conservative static analysis [20, 48, 30]. However, in SDSE, we are always asking about PN-reachability from the current instruction. Hence, rather than solve reachability for an arbitrary initial P-path segment (which would correspond to asking about distances from the current

instruction in all calling contexts of that instruction), we restrict the initial P-path segment to the functions on the current call stack. For performance, we statically pre-compute N-path and S-path distances for all instructions to the target and combine them with P-path distances on demand.

Variants. We consider several variants of SDSE:

SDSE-intra. (Intraprocedural SDSE) In this variant, we ignore call-chains: if the target is not in the current function, then the distance-to-target is ∞. By comparing this variant to regular SDSE, we can see if interprocedurality of distances is crucial for SDSE's effectiveness.

SDSE-pr. (Probabilistic SDSE) In each iteration, this strategy picks a state with probability inversely proportional to the corresponding distance to the target. The rationale is, by randomizing the choice, it is less likely to get stuck in a path which does not lead to the target.

RR(RP,SDSE). (Round-robin of random-path and SDSE) Inspired by KLEE, this strategy alternates between random-path and SDSE, using one strategy for each iteration. We observe that SDSE is much more likely to get stuck in the search than random-path, therefore mixing the two ensures that the search always makes progress. However, It is possible that SDSE might be effective on a program, but RR(RP,SDSE) becomes twice as slow on the same program because only half of the time is spent on SDSE.

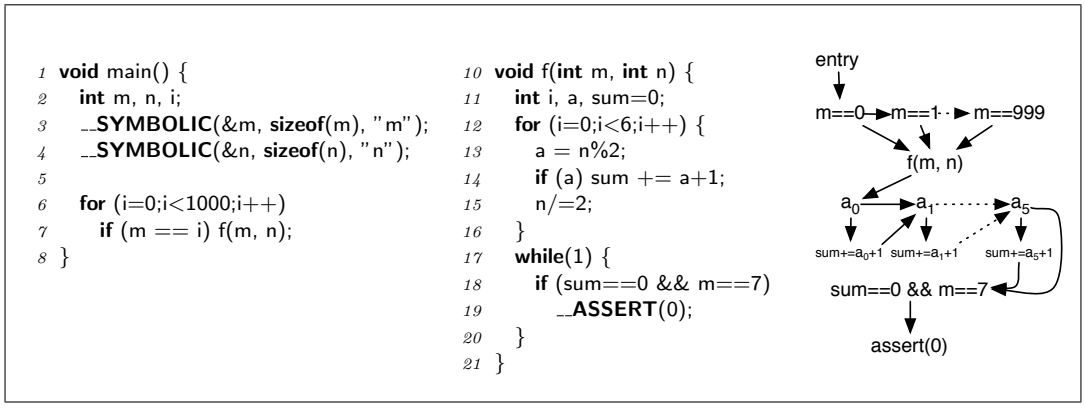


Figure 3.3: Example illustrating CCBSE's potential benefit.

Moreover, we also consider the batched (Section 2.11) versions of all the variants of SDSE. Batching potentially lowers SDSE's overhead, but it can also hurt its effectiveness.

3.1.2 Call-chain-backward symbolic execution

SDSE is often very effective, but there are cases on which it does not do well—in particular, SDSE is less effective when there are many potential paths to the target line, but there are only a few, long paths that are realizable. In these situations, CCBSE can sometimes work better.

To see why, consider the code in Figure 3.3. This program initializes m and n to be symbolic and then loops, calling f(m, n) when m == i for i $\in [0, 1000)$. For non-negative values of n, the loop in lines 12–16 iterates through n's least significant bits (stored in a during iteration), incrementing sum by a+1 for each non-zero a. Finally, if sum == 0 and m == 7, the failing assertion on line 19 is reached. Otherwise, the program falls into an infinite loop, as sum and m are never updated in the loop.

RP, OtterKLEE, OtterSAGE, and SDSE all perform poorly on this example. SDSE gets stuck at the very beginning: in main's for-loop, it immediately steps into f when m == 0, as this is the "fastest" way to reach the assertion inside f according to the ICFG. Unfortunately, the guard of the assertion is never satisfied when m is 0, and therefore SDSE gets stuck in the infinite loop. SAGE is very likely to get stuck, because the chance of SAGE's first generation (Section 2.13.2) entering f with the right argument (m == 7) is extremely low, and SAGE always runs its first generation to completion, and hence will execute the infinite loop forever. RP and OtterKLEE will also reach the assertion very slowly, since they waste time executing f where m$\neq$ 7; none of these paths lead to the assertion failure.

In contrast, CCBSE begins by running f with both parameters m and n set to symbolic, as CCBSE does not know what values might be passed to f. Hence, CCBSE will potentially explore all 2^6 paths induced by the for loop, and one of them, say p, will reach the assertion. When p is found, CCBSE will jump to main and explore various paths that reach the call to f. At the call to f, CCBSE will follow p to short-circuit the evaluation through f (in particular, the 2^6 branches induced by the for-loop), and thus quickly find a realizable path to the failure.

Implementation. CCBSE is implemented in the manage_targets and pick functions from Figure 2.2. Otter states s, returned by pick, include the function f in which symbolic execution started, which we call the *origin function*. Thus, traditional symbolic execution states always have main as their origin function, while CCBSE allows different origin functions. In particular, CCBSE begins by initializing states

for functions containing target lines.

To start symbolic execution at an arbitrary function, Otter must initialize symbolic values for the function's inputs (parameters and global variables). Integer-valued inputs are initialized to symbolic words, and pointers are represented using *conditional pointers* (Section 2.6.3), manipulated using Morris's general axiom of assignment [8, 39]. To support recursive data structures, Otter initializes pointers lazily—we do not actually create conditional pointers until a pointer is used, and we only initialize as much of the memory map as is required. When initialized, pointers are set up as follows: for inputs `p` of type *pointer to type* T, we construct a conditional pointer such that `p` may be null or `p` may point to a fresh symbolic value of type T. If T is a primitive type, we also add a disjunct in which `p` may point to the beginning of an array of 8 fresh values of type T. This last case models parameters that are pointers to arrays, and we restrict its use to primitive types for performance reasons. In our experiments, we have not found this restriction to be problematic.

To illustrate how this strategy for initializing pointers work, consider a pointer `head` of a structure

```
struct node { struct node* next; char* s; } head;
```

When `head` is initialized, it becomes a conditional pointer that is either null or points to a fresh symbolic value of type **struct** `node`. However, `head->next` and `head->s` are uninitialized until they are used. When `head->next` is used later, it is initialized in the same manner as `head` was; and when `head->s` is used, it is initialized to a

```
8  manage_targets (s)
9    (sf,p) = path(s)
10   if pc(p) ∈ targets
11     update_paths(sf, p)
12   else if pc(p) = callto(f) and has_paths(f)
13     for p' ∈ get_paths(f)
14       if (p + p' feasible)
15         update_paths(sf, p + p')
16 update_paths (sf, p)
17   if not(has_paths(sf))
18     add_callers(sf,worklist)
19   add_path(sf, p);
```

Figure 3.4: Target management for CCBSE.

conditional pointer which is either null, a pointer to a fresh symbolic **char**, and a pointer to the beginning of symbolic **char** array of length 8.

Notice that this strategy for initializing pointers is unsound in that CCBSE could miss some targets, but the final paths CCBSE produces are always feasible since they ultimately connect back to main.

The pick function works in two steps. First, it selects the origin function to execute, and then it selects a state with that origin. For the former, it picks the function f with the shortest-length call chain from main. For non-CCBSE the origin will always be main. At the start of CCBSE with a single target, the origin will be the one containing the target; as execution continues there will be more choices—picking the "shortest to main" ensures that we move backward from target functions toward main. After selecting the origin function f, pick chooses one of f's states according to some forward search strategy. We write CCBSE(S) to denote CCBSE using forward search strategy S.

The manage_targets(s) function is given in Figure 3.4. Recall from Figure 2.2 that s has already been added to the worklist for additional, standard forward

search; the job of manage_targets is to record which paths reach the target line and to try to connect s with path suffixes previously found to reach the target. The manage_targets function extracts from s both the origin function sf and the (inter-procedural) *path* p that has been explored from sf to the current point. This path contains all the decisions made by the symbolic executor at condition points. If path p's end (denoted pc(p)) has reached a target (line 10), we associate p with sf by calling update_paths; for the moment one can think of this function as adding p to a list of paths that start at sf and reach targets. Otherwise, if the path's end is at a call to some function f, and f itself has paths to targets, then we may possibly extend p with one or more of those paths. So we retrieve f's paths, and for each one p' we see whether concatenating p to p' (written $p + p'$) produces a feasible path. If so, we add it to sf's paths. Feasibility is checked by attempting to symbolically execute p' starting in p's state s.

Now we turn to the implementation of update_paths. This function simply adds p to sf's paths (line 19), and if sf did not previously have any paths, it will create initial states for each of sf's callers (pre-computed from the call graph) and add these to the worklist (line 17). Because these callers will be closer to main, they will be subsequently favored by pick when it chooses states.

CCBSE(SDSE). When using SDSE as the forward search strategy of CCBSE, we modify SDSE slightly to compute shortest distances to the target line *or* to the functions reached in CCBSE's backward search. This allows SDSE to take better advantage of CCBSE (otherwise it would ignore CCBSE's search in determining

```
1  void main() {
2    int m, n;
3    __SYMBOLIC(&m, sizeof(m), "m");
4    __SYMBOLIC(&n, sizeof(n), "n");
5    foo(); // Some work
6    if (m >= 30) g(m, n);
7  }
8  void g(int m, int n) {
9    int i;
10   for (i=0;i<1000;i++) {
11     baz(); // Some work
12     if (m == i) f(m, n);
13   }
14 }
```

```
15 void f(int m, int n) {
16   int i, a, sum=0;
17   for (i=0;i<6;i++) {
18     a = n%2;
19     if (a) sum += a+1;
20     n/=2;
21   }
22   while (1) {
23     if (sum==0 && m==43)
24       __ASSERT(0);
25   }
26 }
```

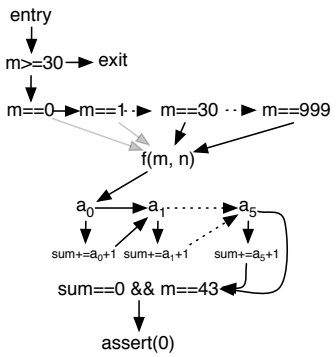


Figure 3.5: Example illustrating Mix-CCBSE's potential benefit.

which paths to take).

3.1.3 Mixing CCBSE with forward search

While CCBSE may find a path more quickly, it comes with a cost: its queries tend to be more complex than in forward search, and it can spend significant time trying paths that start in the middle of the program but are ultimately infeasible. Consider Figure 3.5, a modified version of the code in Figure 3.3. Here, main calls function g, which acts as main did in Figure 3.3, with some m >= 30 (line 6), and the assertion in f is reachable only when m == 43 (line 23). All other strategies fail in the same manner as they do in Figure 3.3.

However, CCBSE also fails to perform well here, as it does not realize that m is at least 30, and therefore considers ultimately infeasible conditions $0 \leq \mathsf{m} < 43$ in f. With Mix-CCBSE, however, we conceptually start forward symbolic execution from main at the same time that CCBSE ("backward search") is run. As before, the backward search will gets stuck in finding a path from g's entry to the assertion.

However, in the forward search, g is called with m $\geq$ 30, and therefore f is always called with m $\geq$ 30, making it hit the right condition m == 43 very soon thereafter. Notice that, in this example, the backward search must find the path from f's entry to the assertion *before* f is called with m == 43 in the forward search in order for the two searches to match up (e.g., there are enough instructions to run in line 5). Should this not happen, Mix-CCBSE degenerates to its constituents running independently in parallel (plus the overhead of manage_targets).

Implementation. We implement Mix-CCBSE with a slight alteration to pick. At each step, Mix-CCBSE decides whether to use regular forward search or CCBSE next, splitting the strategies by time spent, i.e., it switches between the two and maintains a constant ratio between the times spent on them. We tried several ratios (50%, 60% and 75% of the time dedicated to forward search) on our benchmark (Section 3.3), and found that Mix-CCBSE with a ratio of 75% gives the best overall runtime.

3.2 Multi-Target Directed Symbolic Execution

A natural extension to DSE is to generalize it to *multi-target DSE*, which tries to reach multiple targets. More specifically, given a set of line targets, extend DSE to find inputs that drive the program execution to as many targets as possible, within a time limit. One application of this is to improve code coverage. Suppose a program comes with a test suite which achieves a certain coverage. Lines not covered by the test suite are most likely the corner cases where tests are hard or

time-consuming to derive. We can then treat these lines as targets, and use directed symbolic execution to find them. Another similar application is to improve the code coverage of undirected symbolic execution, such as KLEE.

The most straightforward way to carry at multi-target directed symbolic execution is to run (single-target) directed symbolic execution once per line target. But could we do better? In the remainder of this section, we discuss how to extend the idea of SDSE to multi-target DSE.

Multi-Target SDSE. The generalization from SDSE to multiple targets is mostly straightforward. We also add one variant of SDSE that is specific to multi-target. In all variants of SDSE, a target is removed once it is covered, so that strategies will not spend time reaching the same target again.

SDSE. (Shortest distance of all) Pick the state closest to any one of the targets. The goal is to reach *some* target as quickly as possible.

SDSE-pr. Similar to multi-target SDSE, but it picks a state with probability inversely proportional to its shorest distance to any target.

SDSE-rr. (Round-robin of line targets[1]) On the $k|T| + i$th iteration, pick a state that is closest to the ith target of of $|T|$ targets. This can be better than running SDSE in parallel, one for each target, because paths to different targets might share common prefixes.

[1]Not to be confused with round-robin of two strategies.

Just like single-target DSE, we also consider combining SDSE with random-path. We think such a combination is even more meaningful in the context of multi-target DSE, because the multi-target line reachability problem lies between the problems of (1) single-target line reachability and (2) maximizing code coverage (that random-path tends to be good at). In fact, maximizing code coverage is a special case of multi-target line reachability where all uncovered lines are targets.

Apart from combining two strategies using round-robin (just like RR(RP,SDSE) from single-target), we also consider applying strategies *one after another*. The idea is that we can divide the exploration into two *phases*, where in the first phase an undirected strategy is used to find as many targets as possible. When the coverage converges and does not increase for a while, it advances to the second phase, which uses a directed strategy to locate remaining targets. Formally, we define the strategy $\mathrm{Ph}(U, D, r)$:

1. Undirected strategy U is used as long as the following holds: if the ith covered target is reached at time t, then the $(i+1)$th covered target should be reached by time $r \times t$.

2. If it times out waiting for the next covered target, the combined strategy switches to directed strategy D.

When there is only one target, or when no target has been reached by the search, $\mathrm{Ph}(U, D, r)$ degenerates to U. In the experiment we evaluate batched Ph(OtterKLEE,SDSE,3).

Multi-Target CCBSE. We leave the generalization of CCBSE to multiple targets as future work.

3.3 Experiments

3.3.1 Single-Target Directed Symbolic Execution

We evaluated our directed search strategies by comparing their performance to reach the target lines in the small example programs from Section 3.1, and to reach lines that manifest 10 bugs reported in 9 programs from GNU Coreutils version 6.10. These bugs were previously discovered by KLEE [11]. All experiments were run on a machine with six 2.4Ghz quad-core Xeon E7450 processors and 48GB of memory, running 64-bit Linux 2.6.26. We ran 16 tests in parallel, observing minimal resource contention.[2] The tests required less than 4 days of elapsed time. Total memory usage was below 1GB per test.

The results are presented in Table 3.1. Each column represents a strategy, and each row represents a benchmark program. Strategies include

- The SDSE family: SDSE, SDSE-pr and SDSE-intra, and the batched versions of SDSE and SDSE-pr (denoted as B(*)). Also, a strategy RR(RP,SDSE) by round-robin of random-path and SDSE.

- Two variants of CCBSE, using random-path and SDSE as the forward strategy.

[2]This is determined by running a set of tests in different number of parallel jobs, and observing the increase in running time per test.

- Forward strategies (RP, OtterKLEE and OtterSAGE) implemented in Otter, both by themselves and mixing with CCBSE(RP) (running forward strategies 75% of the time). We chose CCBSE(RP) because it was the best overall of the two from part (c), and because RP is the fastest out of the 3 forward-only strategies (RP is more resistant to getting stuck and is inexpensive to compute). Below, we write Mix-CCBSE(S) to denote the mixed strategy where S is the forward search strategy and CCBSE(RP) is the backward strategy.

- The *original* KLEE version r130848 [29].

We did not directly compare against execution synthesis (ESD) [57], a previously proposed directed search strategy; in Section 5.1 we relate our results to those reported in the ESD paper.

We found that the randomness inherent in most search strategies and in the STP theorem prover introduces tremendous variability in the results. Thus, we ran each strategy/target condition 21 times, using integers 1 to 21 as random seeds for Otter. (We were unable to find a similar option in KLEE, and so we simply ran it 21 times.) The main numbers in Table 3.1 are the medians of these runs, and the small numbers are the semi-interquartile range (SIQR). The number of outliers—which fall 3×SIQR below the lower quartile or above the upper quartile, if non-zero—is given in parentheses. We ran each test for at most 900 seconds for the synthetic examples, and at most 1,800 seconds for the Coreutils programs (except for `pr` and `tac`, where tests are given 7,200 seconds to run, as their bugs are more complex than the others). The median is ∞ if more than half the runs timed out, while the SIQR

Program	SDSE	SDSE-pr	SDSE-intra	B(SDSE)	B(SDSE-pr)	RR(RP,SDSE)	B(RR(RP,SDSE))
Figure 3.1	0.2 0.0(5)	0.3 0.0(2)	0.3 0.0(1)	0.3 0.1	0.3 0.1	0.3 0.0(3)	0.3 0.1
Figure 3.3	∞	147.9 24.3(3)	∞	∞	145.4 40.3(3)	371.5 24.6(4)	209.9 31.8(4)
Figure 3.5	∞	∞	∞	∞	∞	∞	∞
mkdir	35.4 ∞	232.4 42.7(3)	424.9 97.2(6)	127.7 5.5(3)	159.4 332.2(5)	∞	∞
mkfifo	23.2 0.8(9)	1,051.5 ∞	∞	22.1 0.7(8)	93.4 414.5(4)	451.6 28.5(3)	258.1 35.5(4)
mknod	∞	∞	∞	∞	∞	∞	1,218.0 285.7(3)
paste	18.9 1.3(3)	57.0 13.2(4)	23.7 2.0(2)	21.6 2.4(1)	25.1 2.0(5)	22.1 2.0(4)	21.5 1.7(3)
seq	574.5 108.4(4)	42.3 43.3(4)	∞	407.6 57.7(6)	41.0 10.0(6)	1,731.4 ∞	674.5 191.8(3)
ptx	439.0 231.2	47.5 522.8(5)	974.8 524.7	31.6 228.3(5)	37.4 4.3(5)	122.9 12.1(4)	97.2 27.1(4)
ptx2	1,729.6 ∞	∞	∞	∞	∞	293.7 28.0(5)	239.8 28.5(5)
md5sum	25.6 1.7(3)	∞	∞	26.9 1.3(5)	∞	31.5 2.7(2)	33.7 2.3(5)
tac	∞	∞	∞	∞	5,824.6 643.8(4)	131.7 7.0(6)	102.0 9.9
pr	953.9 1,410.4(5)	∞	∞	3,943.9 860.8	3,045.0 948.2(2)	∞	∞
Total	12,800.1	21,230.7	24,823.4	15,381.3	14,626.0	13,584.9	11,644.8

(a) SDSE strategies

Program	KLEE
Figure 3.1	0.8 0.0(4)
Figure 3.3	∞
Figure 3.5	∞
mkdir	∞
mkfifo	667.0 269.8(7)
mknod	656.5 ∞
paste	33.8 22.1(3)
seq	51.6 13.9(1)
ptx	313.4 ∞
ptx2	∞
md5sum	∞
tac	∞
pr	∞
Total	21,522.3

(b) KLEE

Program	CCBSE w/ X=		OtterKLEE		OtterSAGE		RP	
	SDSE	RP	Pure	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE
Figure 3.1	0.3 0.1	1.3 0.2(4)	27.9 18.2(4)	23.6 18.2(4)	∞	∞	1.3 0.3(2)	1.4 0.3(2)
Figure 3.3	8.0 0.9(5)	68.1 6.5(1)	407.4 63.9(5)	495.3 53.9(7)	∞	∞	173.1 8.5(7)	246.6 11.8(2)
Figure 3.5	∞	∞	∞	822.3 ∞	∞	∞	∞	363.9 29.1(3)
mkdir	∞	148.4 31.9(4)	199.6 35.8(2)	152.4 32.5(1)	337.1 314.1(4)	389.8 464.9(3)	143.9 9.7(3)	125.8 11.9(1)
mkfifo	25.7 1.2(4)	62.2 13.5	57.9 4.8(5)	46.3 6.9(4)	108.4 79.6(5)	102.1 59.0(5)	58.6 3.2(2)	46.3 4.6(1)
mknod	∞	199.1 59.0	182.2 19.3(3)	122.3 16.9(5)	116.4 154.0(5)	126.5 221.7(5)	205.8 11.6(1)	140.5 9.0(2)
paste	22.8 1.4(4)	27.9 1.2(4)	16.6 0.8(4)	21.8 1.7(3)	17.9 3.6(2)	24.5 13.8(3)	20.1 1.0(5)	27.0 1.8(2)
seq	1,791.9 ∞	407.1 20.1(4)	1,130.6 284.5(5)	138.6 22.0(4)	∞	279.1 ∞	341.7 26.6(3)	180.4 19.2(5)
ptx	1,010.4 520.5	103.8 10.0(1)	100.8 21.2(4)	168.0 27.8(8)	∞	∞	79.0 3.3(6)	130.9 14.8(3)
ptx2	∞	665.1 38.4(8)	735.5 38.4(5)	1,062.3 110.0(4)	∞	∞	399.6 16.0(3)	610.4 77.4
md5sum	36.0 1.1(8)	∞	∞	∞	∞	∞	∞	∞
tac	∞	∞	4,826.6 254.5(4)	6,905.3 ∞	∞	∞	3,165.7 183.2(4)	4,700.6 433.4(5)
pr	∞	∞	∞	∞	5,729.5 ∞	6,462.6 ∞	∞	∞
Total	22,686.8	17,813.5	16,249.7	17,617.1	20,709.2	19,984.6	13,414.5	14,961.9

(c) CCBSE, forward-only and Mix-CCBSE strategies (75% forward)

Table 3.1: Single-target experimental results. For each Coreutils program and for the total, the fastest two times are highlighted. Key: Median SIQR(Outliers) ∞ : time out

is ∞ if more than one quarter of the runs timed out. We highlight the fastest two times in each row.

3.3.1.1 Synthetic programs

The first three rows in Table 3.1 give the results from the examples in Figures 3.1, 3.3, and 3.5. In all cases the programs behaved as predicted.

For the program in Figure 3.1, all the SDSE strategies performed very well. Since the target line is in main, CCBSE(SDSE) is equivalent to SDSE, so it performed equally well. OtterKLEE took much longer to find the target, whereas OtterSAGE timed out in all runs. RP was able to find the target, but it took slightly longer than the SDSEs. Lastly, KLEE performed very well also, although it was still slower than the SDSEs in this example.

For the program in Figure 3.3, CCBSE(SDSE) found the target line quickly, while CCBSE(RP) did so in reasonable amount of time. CCBSE(SDSE) was much more efficient, because with this strategy, after each failing verification of f(m,n) (when $0 \leq$ m < 7), it chose to try f(m+1,n) rather than stepping into f, as f is a target added by CCBSE and is closer from any point in main than the assertion in f is. The remaining strategies took much longer to finish or timed out.

For the program in Figure 3.5, Mix-CCBSE(RP) and Mix-CCBSE(OtterKLEE) performed the best among all strategies, as expected. However, Mix-CCBSE(OtterSAGE) performed far worse. This is because its forward search (SAGE) got stuck in one value of m in the very beginning, and therefore it and the backward search did not

match up. All the remaining strategies timed out.

3.3.1.2 GNU Coreutils

The lower rows of Table 3.1 give the results from the Coreutils programs. The 9 programs we analyzed contain a total of 5.2 kloc and share a common library of about 30 kloc. (There are two bugs in ptx; we name the benchmarks ptx and ptx2 for these bugs.) For each bug, Otter reports a target as being reached when an error (such as buffer overflows and similar errors) occurs at the line target.

The Coreutils programs receive input from the command line and from standard input. We initialized the command line as in KLEE [11]: given a sequence of integers $n_1, n_2, \cdots, n_k$, Otter sets the program to have (excluding the program name) at least 0 and at most k arguments, where the ith argument is a symbolic string of length n_i. All of the programs we analyzed used $(10, 2, 2)$ as the input sequence, except for mknod $(10, 2, 2, 2)$ since its bug requires 4 input arguments to manifest, and ptx2 $(2, 2)$, pr $(2, 1)$ and tac $(2, 2, 2)$, to shorten the time needed to reach the targets. Standard input is modeled as an unbounded stream of symbolic values. Note that Coreutils programs make extensive use of the C standard library, which Otter has to model (Section 2.12).

The last row in Table 3.1 totals the median times for the Coreutils programs for each strategy, counting time-outs as 1,800s (7,200s for pr and tac).

Analysis of SDSEs. We will first look at the SDSE strategies in Table 3.1a. Overall, these strategies (except SDSE-intra) performed very well on many programs. For

example, SDSE and B(SDSE) achieve the best running time on mkdir, mkfifo and md5sum, and close to the best on paste. Interestingly, SDSE-pr and B(SDSE-pr) performed well on another set of programs, seq and ptx, but it did not do well on many others. Optimizing SDSE-pr by batching helped a lot (e.g., batching improves SDSE-pr's runtime on mkfifo from 1,051.5 to 93.4, and decreases its total runtime by 31%). However, batching as a heuristic does not always improve performance. In particular, it increases SDSE's total runtime by 20%. This makes sense, because as discussed in Section 2.11, batching could make Otter spend too much time on unwanted paths. For example, consider the code

```
1 a=1;
2 if(a) /* A lot of work */
3 else /* Target */
```

Both regular and batched SDSEs will direct the search to the conditional. However regular SDSE will stop at the conditional since the false branch is infeasible, while batched SDSE will run over the conditional and follow the true branch, hence wastes time there.

SDSE-intra performed far worse than the other SDSE strategies. This indicates that the inter-procedurality of the distance calculation is crucial for SDSE's effectiveness. (Our benchmark programs have a maximum stack depth of 7.)

We notice that both SDSE and SDSE-pr and their batched variants timed out on mknod. Examining this program, we found it shares a similar structure with mkdir and mkfifo, sketched in Figure 3.6. These programs parse their command line arguments with getopt_long, and then branch depending on those arguments; several of these branches call the same function quote(). In mkdir and mkfifo, the

```
1 int main(int argc, char** argv) {
2     while ((optc = getopt_long (argc, argv, opts, longopts, NULL)) != -1) { ... } ...
3     if (/* some condition */) quote(...);
4     ...
5     if (/* another condition */) quote(...);
6 }
```

Figure 3.6: Code pattern in mkdir, mkfifo and mknod

target is reachable within the first call to quote(), and thus SDSE can find it quickly. However, in mknod, the bug is only reachable in a later call to quote()—but since the first call to quote() is a shorter path to the target line, SDSE takes that call and then gets stuck inside quote(), never returning to main() to find the path to the failing assertion. SDSE and SDSE-pr and their batched variants also failed entirely on ptx2.

Given that a pure SDSE strategy can get stuck in the search easily, we were eager to try RR(RP,SDSE), the strategy of round-robining random-path and SDSE. We found that the batched version of this strategy, B(RR(RP,SDSE)), gives the best overall results—it achieved the best total time of 11,644.8s—among all directed and undirected strategies. However, in many cases it did not achieve the best performance per program. For example, it timed out in mkdir, but its constituents did not (B(RR(RP,SDSE)) actually returned in 9 out of 21 runs; see Figure A.6 for its beeswarm plot). Also, in mkfifo, B(RR(RP,SDSE)) ran for 251.8s, which is longer than both of its constituents (SDSE: 23.2s; RP: 58.6s). These show that two strategies, when combined in a round-robin fashion, can affect each other and ruin both's effectiveness. Lastly, B(RR(RP,SDSE)) did not finish in pr, likely because

random-path did not finish either.

Analysis of CCBSEs and Mix-CCBSEs. CCBSEs performed less well. CCBSE(SDSE) timed out on many programs, while CCBSE(RP) timed out on the last 3 programs, although its performance on the remaining programs is not impressive. This is not too surprising, because we expect CCBSE will impose too much overhead when running on its own.

On the other hand, Mix-CCBSEs performed a lot better (except for Mix-CCBSE(OtterSAGE), possibly because OtterSAGE was ineffective on some programs). In our prior work on DSE [36] in which the benchmark suite consisted of only the first 6 Coreutils programs of Table 3.1a. we showed that Mix-CCBSE(OtterKLEE) was the best strategy in terms of total runtime. While the overall result of these programs is the same, our implementation has changed. In particular:

- We now use a different algorithm for splitting between forward search and CCBSE in Mix-CCBSE. Splitting requires measuring time. Our prior work did not use wallclock time, but instead a "system time" defined as a weighted sum (50 × number of STP queries + number of steps made by Otter) in favor of experiment reproducibility. And it split the system time *equally* between the forward search and CCBSE. However, we later found that system time did not split searches evenly in wallclock time for some programs under the new version of Otter, and so we abandoned it. In the experiment, Mix-CCBSE spends 75% of wallclock time on the forward search.

- We now use a different version of STP (r1377 versus r1213 used in our prior

work). Notice that CCBSE/Mix-CCBSE make more complex queries in general;

- The experimental setup is different. In our prior work programs were slightly modified so that bugs were marked explicitly as **assert**(0); now we specify the line targets and Otter tracks errors that occur in these lines.

Nevertheless, our results do show that mixing a forward search with CCBSE can give a significant improvement in some cases—for OtterKLEE and random-path, the total times are notably less when mixed with CCBSE. This is true for Mix-CCBSE(OtterKLEE): it ran dramatically faster on seq than either of its constituents (138.6s for the combination versus 1,130.6s for OtterKLEE and 1,791.9s for CCBSE(RP)), and on mknod (122.3s for the combination versus 182.2s for OtterKLEE and 199.1s for CCBSE(RP)). The case on mknod demonstrates the benefit of mixing forward and backward search: in the combination, CCBSE(RP) found the failing path inside of quote() (recall Figure 3.6), and OtterKLEE found the path from the beginning of main() to the right call to quote().

On the other hand, Mix-CCBSEs took a long time to finish or timed out in the last 4 programs. This is because their constituents (in particular the undirected forward strategies) did not do well either.

Summary. Overall, batched RR(RP,SDSE) has the fastest total running time across all strategies, and although it is not the fastest search strategy per program, it is subjectively fast enough on these examples. Thus, our results suggest that the best single strategy option for solving line reachability is batched RR(RP,SDSE).

3.3.2 Multi-Target Directed Symbolic Execution

We use a similar experimental setup from the single-target DSE in our multi-target DSE. In particular, for each program we use the same configuration (e.g, symbolic input size) as in single-target. We also run each (program, strategy) pair 5 times with different random seeds and observe the statistical variance. Here are the differences:

Choice of targets. We pick our targets by running (natively) Coreutils' test suite and picking all the uncovered lines of the programs, according to gcov, a code coverage analysis tool from GNU. For simplicity, we exclude uncovered code from Coreutils' library (e.g., for `mkdir` we only consider lines in `mkdir.c`).

Time limit. We set a fixed time limit of 2 hours for each test. We try to see how many lines a strategy covers given a time period. (As a consequence, the multi-target experiment takes much longer to run, and therefore we could not run on our shared benchmarking machine with as many seeds as in the single-target experiment.)

Upon reaching a target. When a line target is reached, the execution at that point does not stop (unless the target triggers an error). Instead, the reached line target is removed from the set of uncovered targets and the execution keeps going.

Strategies. For SDSE strategies, we only consider the batched versions. We anticipate that batching is likely to help, because longer running time results

in more states waiting in Otter's scheduler, thereby increasing the time to compute SDSE's heuristic decision.

The multi-target SDSE strategies discussed in Section 3.2 are added for comparison. We also include B(RR(RP,SDSE-rr)) in our experiment, however not B(RR(RP,SDSE-pr)), as B(SDSE-pr)'s evaluation is poor (Table 3.2). SDSE-intra and CCBSE(SDSE) are removed since they are shown to be ineffective from the single-target experiment. We leave the comparison with the original KLEE as future work.

The experiment was run on the same machine as the single-target experiment, and the same number (16) of tests in parallel. The tests required less than 3 days of elapsed time.

Results and Discussions. The results of the multi-target experiment are shown in Table 3.2. For each program, its number of targets is shown in parentheses next to it. Each program has two rows, one showing the coverage (number of lines covered) and another showing the time taken to cover that many lines. Similar to Table 3.1, an entry in the table shows the median, SIQR and number of outliers out of a series of runs, although we only ran each test 5 times, using seeds from 1 to 5.

The last two lines in Table 3.2 the totals. The two notions of totals we use are

Average coverage (Avg%). This is the average of (number of covered targets)/(number of targets) over all programs, which has the benefit that no program will dominate the result because it has a lot of targets (e.g., ptx).

Program		B(SDSE)	B(SDSE-pr)	B(SDSE-rr)	B(RR(RP, SDSE))	B(RR(RP, SDSE-rr))	B(Ph(OtterKLEE, SDSE,3))
mkdir(8)	Cov.	5 0	5 0	5 0	8 0	8 0	7 0
	Time	32.8 2.0(1)	43.1 4.2(2)	34.7 2.2(1)	255.1 20.0(1)	232.9 20.8(1)	35.1 2.3(2)
mkfifo(11)	Cov.	11 0	9 1	11 0	11 0	11 0	9 1
	Time	179.3 5.1(1)	1,476.2 1,728.1	240.9 12.3(1)	191.6 8.8(1)	141.8 10.5(1)	43.3 77.4(1)
mknod(23)	Cov.	22 0	14 0	21 0	23 0	23 0	15 1
	Time	1,947.1 183.1(1)	1,528.6 500.3(1)	4,330.3 246.4(1)	310.3 22.2(1)	212.3 25.4(1)	740.2 178.3
paste(78)	Cov.	32 0	32 1	32 9	58 0	60 0	62 0
	Time	437.7 75.7(2)	3,701.9 3,039.2	596.2 258.0(2)	2,614.8 108.3(2)	3,922.1 96.9(2)	888.7 344.2(1)
seq(16)	Cov.	8 0	7 0	8 0	10 0	10 0	8 1
	Time	413.1 26.0(1)	46.1 8.3	43.7 3,409.1(1)	398.0 62.3	44.3 3.3(1)	141.8 19.1(1)
ptx(517)	Cov.	109 37(1)	109 2(1)	226 58	237 2(1)	222 2(1)	217 2(1)
	Time	124.2 47.3(1)	4,560.4 403.1(2)	1,965.8 842.1	6,314.5 624.8(1)	6,763.5 207.9(2)	6,384.8 180.3
md5sum(65)	Cov.	10 0	13 0	19 0	18 0	18 0	3 1(1)
	Time	4,612.8 543.4(1)	3,962.4 1,719.2	5,020.6 392.9(1)	4,120.4 211.6(1)	3,974.7 731.7(1)	42.4 7.0(1)
tac(51)	Cov.	6 0	5 0	6 0	6 0	6 0	6 0
	Time	3,893.1 359.1	1,654.0 140.5(1)	788.8 47.8(1)	491.5 53.5(1)	668.3 33.6(2)	436.6 181.0(1)
pr(92)	Cov.	64 0(1)	61 0	64 0	44 0	42 0	33 15
	Time	5,250.0 1,008.1	2,072.9 186.2(2)	4,510.9 896.7	5,241.4 875.4	4,706.8 38.3(2)	5,812.4 531.6(1)
	Avg %	51.9	45.2	55.5	63.3	63.1	50.9
	Agg %	31.0	29.6	45.5	48.2	46.5	41.8

(a) SDSE strategies

Program		CCBSE(RP)	OtterKLEE		OtterSAGE		RP	
			Pure	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE
mkdir(8)	Cov.	8 0	8 0	8 0	8 0	8 0	8 0	8 0
	Time	93.6 1.8(2)	96.3 8.5(1)	133.8 11.0(2)	395.9 66.1(1)	376.5 332.2	79.3 0.5(2)	106.2 7.3(1)
mkfifo(11)	Cov.	11 0	11 0	11 0	11 0	11 0	11 0	11 0
	Time	306.2 33.7	534.5 95.1	520.0 183.1(1)	240.4 132.7	458.0 39.1(2)	272.2 3.4(2)	372.6 19.6(1)
mknod(23)	Cov.	22 0	22 0	22 0	18 1	21 2	22 0	22 0
	Time	797.3 48.9	959.4 231.8	1,039.1 167.5(2)	272.8 31.2(2)	435.8 20.9(2)	753.1 25.3(2)	1,073.4 55.1(2)
paste(78)	Cov.	62 1	60 1	62 0(1)	60 0(2)	62 1(1)	62 0(1)	62 0
	Time	6,303.9 2,429.3	3,296.0 350.4(2)	5,712.6 517.1(1)	271.1 74.4(1)	3,987.3 52.0(1)	5,314.1 374.7(2)	3,731.6 413.5(1)
seq(16)	Cov.	10 0	9 0	10 0	8 1(1)	7 0(1)	9 0(1)	10 0
	Time	302.6 17.4(2)	851.1 490.4(1)	1,453.5 545.3	59.6 53.3(1)	6,265.1 3,255.1	257.2 13.5(2)	333.0 35.9(1)
ptx(517)	Cov.	246 2(1)	225 1(1)	230 4	248 13(1)	267 10(2)	246 1	251 0(2)
	Time	5,621.9 449.9(1)	6,458.0 218.1(2)	7,000.8 218.5(1)	6,316.2 74.5(2)	7,037.0 114.1(1)	6,914.4 188.1(1)	6,191.8 56.0(2)
md5sum(65)	Cov.	14 0	15 0(1)	16 0(1)	5 1(2)	13 0(2)	13 0	14 0
	Time	1,188.5 90.6	56.6 234.7(1)	65.6 9.0(1)	36.9 11.0(1)	2,212.3 343.8(2)	1,581.0 392.5(1)	671.4 52.3(1)
tac(51)	Cov.	7 0	6 0	8 0	6 0	8 0(1)	6 0	8 0(1)
	Time	776.4 7.3(2)	280.9 8.0(1)	5,288.9 312.1(1)	276.1 156.4(1)	6,535.0 492.1(1)	646.4 94.8	5,974.2 513.5(1)
pr(92)	Cov.	36 0(1)	35 1	38 0(1)	60 3(1)	33 5(1)	35 7(1)	37 2
	Time	3,440.6 226.0(1)	5,866.0 619.8	6,337.3 660.5(1)	5,734.0 327.3(1)	6,321.5 1,026.6(1)	3,407.1 1,515.9(1)	6,632.6 1,632.6(1)
	Avg %	62.2	60.6	62.6	59.8	59.7	61.0	62.6
	Agg %	48.3	45.4	47.0	49.2	49.9	47.9	49.1

(b) CCBSE, forward-only strategies and their mixes with CCBSE(RP)

Table 3.2: Multi-target experimental results. For each Coreutils program, average and aggregated percentages, the best three strategies are highlighted. Key: Median SIQR(Outliers) ∞ : time out

Aggregated coverage (Agg%). This is (number of covered targets of all programs)/(number of targets of all programs).

We can see from Table 3.2 that most strategies have good coverages on average. Except for B(SDSE), B(SDSE-pr) and B(Ph(OtterKLEE,SDSE,3)), other strategies have at least 55% of average coverage/45% of aggregated coverage, although the magnitude of the differences among them is not large enough for us to draw any conclusions.

Looking at how quickly each strategy covers those lines, we do see more significant differences, however. Figure 3.7 shows the *normalized* coverage-over-time for different strategies. The plot summarizes the per-program coverage-over-time plots shown in Appendix A.2. To create Figure 3.7, we aggregate coverage data as if the 9 Coreutils programs were run in parallel. And, whenever a strategy finds a line target in program P at time t (median of 5 runs), its coverage increases by 1/|number of targets in P| at time t. Notice that 9 is the maximum normalized coverage level. Also, a coverage level divided by 9 gives the average coverage (Avg%) as defined above.

From Figure 3.7 we observe the following. Firstly, undirected strategies RP and OtterKLEE (and B(Ph(OtterKLEE,SDSE,3)) since it begins by running as OtterKLEE) cover quickly in the beginning, until at the coverage level of 3, B(RR(RP,SDSE-rr)) begins to catch up, and it remains the fastest strategy towards the end (when the 2-hour time limit has reached). Also, several directed strategies (B(RR(RP,SDSE)), Mix-CCBSE(RP), Mix-CCBSE(OtterKLEE) and CCBSE(RP)) perform better than

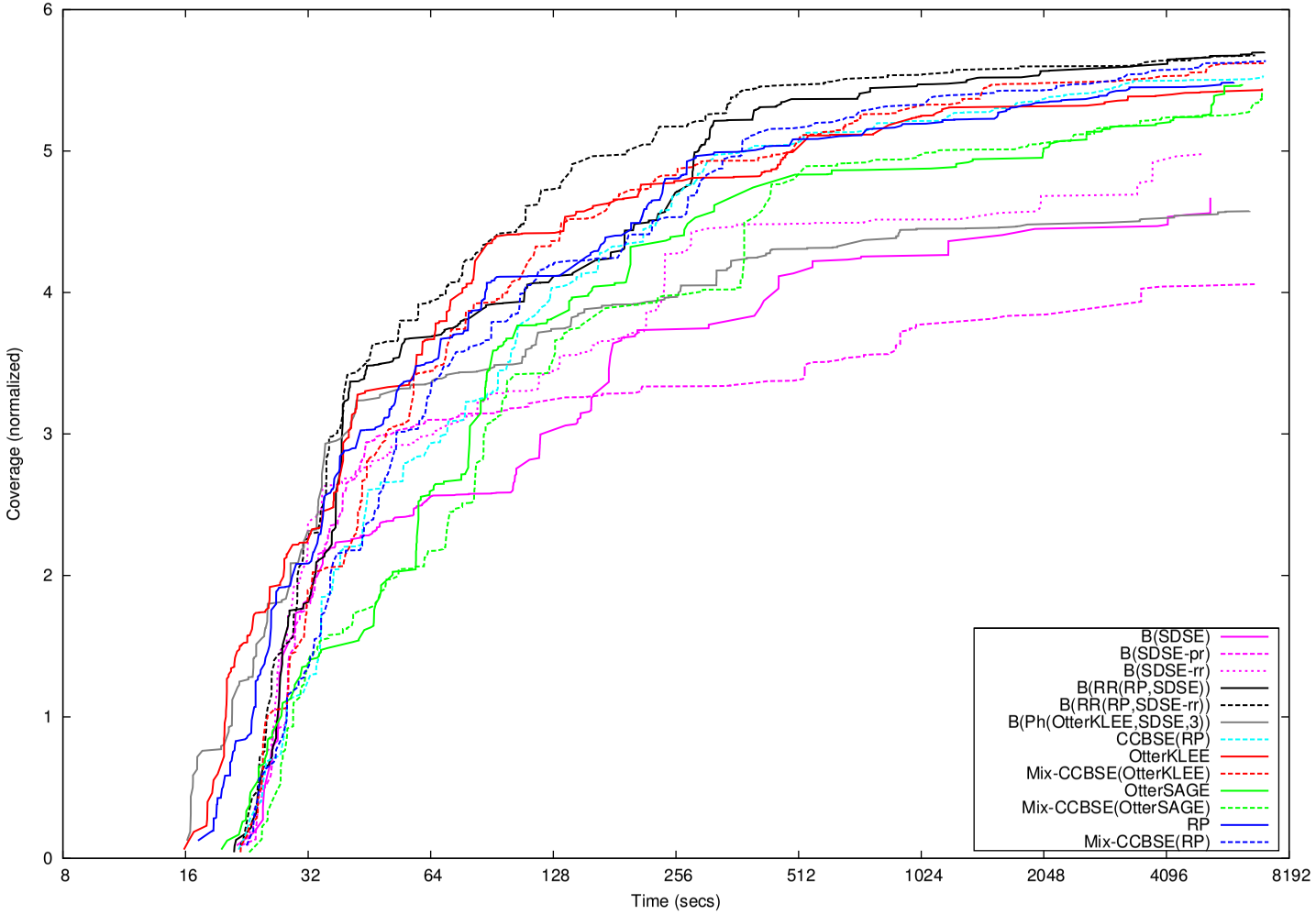


Figure 3.7: Normalized coverage over time. Full coverage is 9.

the undirected strategies since $t = 256$s. Such observation matches our expectation: at the beginning when there are many targets, undirected stategies can get lucky and find them quickly since they have less overhead. However, once the "easy" targets are covered, directed strategies are more effective in covering the remainder. That said, we also observe that pure SDSE strategies performed poorly. We speculate that this is because SDSEs alone can get stuck more easily. We think this also explains why B(Ph(OtterKLEE,SDSE,3)) did not perform well, since it runs SDSE in its second phase.

Hence, we conclude that B(RR(RP,SDSE-rr)) is the best solution for the multi-target line reachability problem, while many other directed strategies are also good candidates. We also conjecture that B(Ph(OtterKLEE,RR(RP,SDSE-rr),3)) performs ever better, combining the good characteristics of its constituents OtterKLEE and B(RR(RP,SDSE-rr)). We leave the proof of our conjecture as future work.

3.3.3 Threats to validity

There are several threats to the validity of our results. First, we were surprised by the wide variability in our running times: the SIQR can be very large—in some cases for CCBSE(SDSE), OtterKLEE and OtterSAGE, the SIQR exceeds the median—and there are many outliers.[3] This indicates the results are not normally distributed, and suggests that randomness in symbolic execution can greatly perturb the results. To our knowledge, this kind of significant variability has not been reported well in the literature, and we recommend that future efforts on symbolic

[3] See Appendix A.1 for beeswarm distribution plots for each cell in the table [36].

execution carefully consider it in their analyses.

Second, our implementation of OtterKLEE and OtterSAGE unavoidably differs from the original versions. The original KLEE is based on LLVM [34], whereas Otter is based on CIL, and therefore they compute distance metrics over different control-flow graphs. Also, Otter uses newlib [41] as the standard C library, while KLEE uses uclibc [54]. These may explain some of the difference between OtterKLEE and the original KLEE's performance in Table 3.1.

Furthermore, the original SAGE is a concolic executor, which runs programs to completion using the underlying operating system, while Otter's implementation of SAGE emulates the run-to-completion behavior by not switching away from the currently executing path. There are other differences between SAGE and Otter, e.g., SAGE only invokes the theorem prover at the end of path exploration, whereas Otter invokes the theorem prover at every conditional along the path. Also, SAGE suffers from *divergences*, where a generated input may not follow a predicted path (possibly repeating a previously explored path) due to mismatches between the system model and the underlying system. Otter does not suffer from divergences because it uses a purely symbolic system model. These differences may make the SAGE strategy less suited to Otter.

Moreover, our conclusions are certainly limited by our choice of benchmark. For instance, we concluded in our prior work that Mix-CCBSE(OtterKLEE) was the best strategy from the first six programs of Table 3.1, but we have a different conclusion after looking at more programs. We leave expanding the benchmark suite as future work. Possible candidates of benchmark programs are the ones studied by

the symbolic execution community, such as BusyBox [1] studied by KLEE.

Finally, we did not obtain as many samples for the multi-target experiment as for the single-target experiment. Our conclusion on multi-target DSE is therefore subject to the high variability of symbolic execution (however, we anticipate that variability is less of an issue for multi-target DSE, since runs are given much longer time).

Chapter 4

Using Symbolic Execution to Understand Behavior in Configurable Software Systems

Otter can be used not just to generate tests, but more generally to exhaustively explore *all* program executions. In this chapter, we present a study that uses this capability to efficiently enumerate all paths enabled by the program's *configuration options*—program inputs that switch on/off different program features. For each path, Otter tracks code coverage (line, basic block, edge, and condition coverage metrics) the path achieves. Then, from this data, we can derive useful information, including guaranteed coverage (Section 4.3) and minimal covering sets (Section 4.7.3), that enhance our understanding of configurable software systems and how best to evaluate them.

Otter generates tens or hundreds of thousands of paths depending on the application, but these are just a small fraction of the tens of millions or more runs that would have been needed had we naively enumerated and tested all configurations. Otter performs this task better than prior related work. Specifically, concolic testers duplicate work executing common prefixes of paths; KLEE's use of LLVM to transform C programs to bytecodes may confuse coverage tracking at source code level; and JPF-SE appears unable to scale to software systems of the size of those we studied.

The remainder of this chapter presents our report of the study previously published [49]; this chapter includes figures and discussions extracted from that publication.

Contributions. This chapter presents joint work with the authors of [49]. In this work, I was in charge of Otter's implementation and the analysis of ngIRCd, one of the subject programs in our study.

4.1 Motivation for the Study

Modern software systems include numerous user-configurable options. For example, network servers typically let users configure the active port, the maximum number of connections, what commands are available, and so on. While this flexibility helps make software systems extensible, portable, and achieve good quality of service, it can also generate a huge number of configurations—in the worst case every combination of option settings is a distinct configuration. This *software configuration space explosion problem* presents real challenges to software developers. It significantly magnifies testing obligations; it makes static analysis much more difficult, as different configurations can be conflated together; and it generally complicates program understanding tasks. All of this adds to development costs.

We conjecture that at certain levels of abstraction, the software configuration space is much smaller than combinatorics might suggest. For example, consider a web server that can be configured to support sequential or concurrent connections and to enable or disable logging. In this case, the block coverage achieved by all four

possible configurations might be exactly the same as that achieved by two configurations: sequential connections with logging enabled, and concurrent connections with logging enabled. (Disabling logging would be unlikely to cover any new blocks.) Thus when considering block coverage, the effective configuration space for our example is half the size we would expect. If our conjecture proves true, then in future work, new techniques and heuristics might be created to partition configuration spaces in ways that greatly simplify testing, analysis, and program understanding.

To evaluate our conjecture, we studied three configurable subject programs: vsftpd, ngIRCd, and grep. For each system, we first identified a sizable number of run-time configuration options to analyze, determined their possible settings, and created a test suite. Then, we marked the selected configuration options as symbolic, and we used Otter to enumerate *all* possible program paths for *all* possible settings of the selected configuration options. We next projected the paths onto four types of structural coverage—line, basic block, edge, and condition coverage—and used the resulting data to discover *interactions* among configuration options. We formally define interactions based on *guaranteed coverage*, which will be discussed in Section 4.3.

4.2 Configurable Software Systems

For the purpose of this study, a configurable system is a generic code base and a set of mechanisms for implementing pre-planned variations in the code base's structure and behavior. Here, we are focusing on run-time configuration options,

```
... else if (tunable_pasv_enable &&
             str_equal_text(&p_sess->ftp_cmd_str, "EPSV"))
{
  handle_pasv(p_sess, 1);
}
... else if (tunable_write_enable &&
             (tunable_anon_mkdir_write_enable ||
              !p_sess->is_anonymous) &&
             (str_equal_text(&p_sess->ftp_cmd_str, "MKD") ||
              str_equal_text(&p_sess->ftp_cmd_str, "XMKD")))
{
  handle_mkd(p_sess);
}
```

(a) Boolean configuration options (vsftpd)

```
else if(Conf_OperCanMode) {
  /* IRC-Operators can use MODE as well */
  if (Client_OperByMe(Origin)) {
    modeok = true;
    if (Conf_OperServerMode)
      use_servermode = true; /* Change Origin to Server */
  }
}
...
  if (use_servermode)
    Origin = Client_ThisServer();
```

(b) Nested conditionals (ngIRCd)

Figure 4.1: Example uses of configuration options (bolded) in subjects.

which are usually given values via configuration files or command-line parameters. A *configuration* is a mapping of configuration options to their settings.

Figure 4.1 illustrates several ways that run-time configuration options can be used, and explains why understanding their usage requires technology such as symbolic execution. All of these examples come from our subject programs. In this figure, variables containing configuration options are shown in boldface.

The example in Figure 4.1a shows a section of vsftpd's command loop, which

receives a command and then uses a long sequence of conditionals to interpret the command and carry out the appropriate action. The example shows two such conditionals that also depend on configuration options (all of which begin with tunable_ in vsftpd). In this case, the configuration options enable certain commands, and the enabling condition can either be simply the current setting of the option (as on line 1) or may involve an interaction between multiple options (as on lines 6–7).

Figure 4.1b shows a different example in which two configuration options are tested in nested conditionals. This illustrates that it is insufficient to look at tests of configuration options in isolation; we also need to understand how they may interact based on the program's structure. Moreover, in this example, if both options are enabled then use_servermode is set on line 24, and its value is then tested on line 28. This shows that the values of configuration options can be indirectly carried through the state of the program.

4.3 Guaranteed Coverage

While Otter generates useful per-path coverage data, this data tells us only a little about our subject programs unless we further analyze it. By definition, each path explored for a particular test case is distinct from all the other paths for the same test case. Thus with no abstraction, every configuration option combination given by a path is unique.

For example, consider the program in Figure 4.2a. This program includes boolean input variables a, b, c, d, and **input**. The first four are intended to represent

```
1  int a=α, b=β, c=γ, d=δ; /* symbolic */
2  int input = /* concrete */;
3  int x = 0;
4  if (a)
5    /* Statement 1 */
6  else if (b) {
7    /* Statement 2 */
8    x = 1;
9    if (!input) {
10     /* Statement 3 */
11   }
12 }
13 int y = c || d;
14 if (x && input) {
15   /* Statement 4 */
16   if (y)
17     /* Statement 5 */
18 }
```

(a) Example program

input=1	Settings	Coverage
s_1	α	$\{1\}$
s_2	$\neg\alpha \wedge \beta \wedge (\gamma \vee \delta)$	$\{2, 4, 5\}$
s_3	$\neg\alpha \wedge \beta \wedge \neg(\gamma \vee \delta)$	$\{2, 4\}$
s_4	$\neg\alpha \wedge \neg\beta$	$\emptyset$

input=0	Settings	Coverage
s_5	α	$\{1\}$
s_6	$\neg\alpha \wedge \beta$	$\{2, 3\}$
s_7	$\neg\alpha \wedge \neg\beta$	$\emptyset$

(b) Option settings and their coverages

Figure 4.2: Example symbolic execution.

run-time configuration options, and so we initialize them on line 1 with symbolic values α, β, γ, and δ, respectively (via __SYMBOLIC (not shown)). The last variable, **input**, is intended to represent program inputs other than configuration options. Thus we leave it as *concrete*, and it must be supplied by the user (e.g., as part of argv (not shown)). We have also indicated five statements, numbered 1–5, whose coverage we are interested in. (We focus on line coverage here for illustration purposes, but the idea is the same for other forms of coverage.)

Figure 4.2b shows, for each input value, distinct settings of the configuration options (represented by constraints on the boolean values of the options). Each setting corresponds to a distinct execution path, and the set of statements covered by the path is also shown. Thus far, we only know that there are, for example, four

distinct paths (for settings s_1, s_2, s_3 and s_4) if **input**=1, and that is fewer than the 16 paths we might naively expect. However, if we are interested in more abstract properties of the program, then paths are no longer unique, and the configuration space collapses further. For example, suppose we are only interested in covering statement 2 in Figure 4.2a. Then we can see that settings s_1 and s_4 are irrelevant, and either settings s_2 or s_3 is sufficient.

For this study, we project paths enumerated by Otter onto four commonly used abstractions of program behavior: line, block, edge, and condition coverage. The principal tool we use to relate configuration options to coverage is *guaranteed coverage*.

Definition 1 *Given a particular coverage criterion, we say that a predicate p over the configuration options* guarantees coverage *(line, block, edge, condition, etc.) of X if there exists some test case such that any configuration satisfying p is guaranteed to cover X.*

For example, from Figure 4.2a we can see that any configuration satisfying $\alpha = 0 \wedge \beta = 1$ (i.e., a=0, b=1) is guaranteed to cover statement 2, no matter the choice of γ and δ.

We can use Otter's per-path coverage to compute the guaranteed coverage for a predicate p, which we will write $Cov(p)$. We first find $Cov^T(p)$, the coverage guaranteed under p by test case T, for each test case; then, $Cov(p) = \bigcup_T Cov^T(p)$. To compute $Cov^T(p)$, let p_i^T be the path conditions from T's symbolic execution, and let $C^T(p_i^T)$ be the covered lines (or blocks, edges, conditions, etc.) that occur

in that path. Then $Cov^T(p)$ is

$$\begin{aligned} Consistent^T(p) &= \{p_i^T \mid SAT(p_i^T \wedge p)\} \\ Cov^T(p) &= \bigcap_{q \in Consistent^T(p)} C^T(q) \end{aligned}$$

In words, first we compute the set of predicates p_i^T such that p and p_i^T are consistent. If this holds for p_i^T, the items in $C^T(p_i^T)$ *may* be covered if p is true. Since Otter explores all possible program paths, the intersection of these sets for all such p_i^T is the set guaranteed to be covered if p is true.

Next, we define *interactions* among options using guaranteed coverage.

Definition 2 *An* interaction *is a set p of option settings that guarantees coverage that is not guaranteed by any subset of p. Moreover, the* strength *of an interaction is the number of option settings it contains.*

For example, since $Cov(\neg\alpha \wedge \beta)$ is a strict superset of $Cov(\neg\alpha) \cup Cov(\beta)$, $\neg\alpha \wedge \beta$ is an interaction. Informally, interactions indicate combinations of options that are "interesting" because they guarantee some new amount of coverage. Moreover, $\neg\alpha \wedge \beta$ has strength 2. Lower-strength interactions place fewer requirements on configurations, whereas higher-strength interactions require more options to be set in particular ways to achieve their coverage.

Table 4.1 lists some predicates, the coverage (set of statements) they guarantee, and whether options within these predicates form an interaction. Note that we cannot guarantee covering statement 5 without setting three symbolic values (although we could have picked δ instead of γ).

p	$Consistent(p)$ (input $= 1$)	$Consistent(p)$ (input $= 0$)	$Cov(p)$	$\bigcup_{p' \subset p} Cov(p')$	Interaction? (Strength?)
α	s_1	s_5	$\{1\}$	$\emptyset$	Yes (1)
$\neg\alpha$	s_2, s_3, s_4	s_6, s_7	$\emptyset$	$\emptyset$	No
β	s_1, s_2, s_3	s_5, s_6	$\emptyset$	$\emptyset$	No
γ	s_1, s_2, s_4	s_5, s_6, s_7	$\emptyset$	$\emptyset$	No
$\alpha \wedge \beta$	s_1	s_5	$\{1\}$	$\{1\}$	No
$\neg\alpha \wedge \beta$	s_2, s_3	s_6	$\{2, 3, 4\}$	$\emptyset$	Yes (2)
$\beta \wedge \gamma$	s_1, s_2	s_5, s_6	$\emptyset$	$\emptyset$	No
$\neg\alpha \wedge \gamma$	s_2, s_4	s_6, s_7	$\emptyset$	$\emptyset$	No
$\neg\alpha \wedge \beta \wedge \gamma$	s_2	s_6	$\{2, 3, 4, 5\}$	$\{2, 3, 4\}$	Yes (3)

Table 4.1: Guaranteed coverage of different predicates, and if options within these predicates form an interaction.

4.4 Tracking Coverage in Otter

The precise definitions of different coverage metrics (line, block, edge, and condition coverage) demand some elaboration, because Otter runs on CIL's representation of the input program, which is simplified to use only a subset of the full C language.

Line coverage. Otter records which CIL statements it executes and projects that back to the original source lines using a mapping maintained by CIL.

Block and edge coverage. Otter groups CIL statements into basic blocks, which are sequences of statements that start at a function entry or a join point; do not contain any join point after the first statement; end in a function call,

`return`, `goto`, or conditional; or fall through to a join point.

Normally, CIL expands short-circuiting logical operators `&&` and `||` into sequences of branches. However, for block and edge coverage, we disable that expansion as long as the right operand has no side effect, so that both operands are computed in the same basic block. Then to compute block coverage, we record which basic blocks are executed, and to compute edge coverage, we compute which control-flow edges between basic blocks are traversed.

Condition coverage. Otter enables expansion of `&&` and `||`, so that each part of a compound condition is always in its own basic block. Otter then computes how many conditions—that is, how many branches—are taken in the expanded program.

4.5 Subject Programs

The subject programs for our study are vsftpd, a widely-used secure FTP daemon; ngIRCd, the "next generation IRC daemon"; and GNU grep, a popular text search utility. All of our subject programs are written in C. Each has multiple configuration options that can be set either in system configuration files or through command-line parameters.

Table 4.2 gives descriptive statistics for each subject program. The top two rows list the program version numbers and lines of code as computed by `sloccount`. The next group of rows lists the number of executable lines, basic blocks, edges, and conditions; these four metrics are what we measure code coverage against, and they

	vsftpd	ngIRCd	grep
Version	2.0.7	0.12.0	2.4.2
# Lines (sloccount)	10,482	13,601	9,124
# Lines (executable)	4,112	4,387	3,302
# Basic blocks	4,584	6,742	5,033
# Edges	5,033	7,374	6,332
# Conditions	2,528	3,432	4,094
# Test cases	64	142	113
# Analyzed conf. opts.	30	13	18
Boolean	20	5	14
Integer	10	8	4
# Excluded conf. opts.	65	16	4

Table 4.2: Subject program statistics.

are based on the CIL representation of the program, as discussed in Section 4.4. To get more accurate measurements, we removed some unreachable code before passing the sources to CIL. Specifically, we commented out 4 unreachable functions in grep. We also forced vsftpd to run in single-process mode, as Otter does not support multiprocess symbolic execution, and correspondingly eliminated 3 files in vsftpd that are reachable only in two-process mode.

One thing to note is that there are more basic blocks than executable lines of code in all 3 programs. This occurs because, in many cases, single lines form multiple blocks. For example, a line that contains a for loop will have at least two blocks (for the initializer and the guard), and lines with multiple function calls are broken into separate blocks according to our definition.

The next row in Table 4.2 lists the number of test cases. In creating these

test cases, we attempted to both cover the major functionality of the system and to maximize overall line coverage. We stopped creating new tests when the remaining uncovered code was overwhelmingly devoted to handling low-level system errors such as malloc() or device read() failures.

vsftpd does not come with its own test suite, so we developed tests to exercise its major functionality such as logging in; listing, downloading, and renaming files; asking for system information; and handling invalid commands.

ngIRCd also does not come with its own test suite, so we created tests based on the IRC functionality defined in RFCs 1459, 2812 and 2813. Our tests cover most of the client-server commands (e.g., client registration, channel join/part, messaging and queries) and a few of the server-server commands (e.g., connection establishment, state exchange), with both valid and invalid inputs.

Grep comes with a test suite consisting of hundreds of tests. To build our test suite for this study, we ran all the test cases in Otter to determine their line coverage. Then, without sacrificing total line coverage, we selected 70 test cases from the original suite for our study. Next, we created 43 new test cases to improve overall line coverage. The final analysis was done using these 113 test cases.

Finally, the last group of rows in Table 4.2 counts the configuration options. We give the total number of analyzed configuration options, i.e., those that we treated as symbolic, and also break them down by type (boolean or integer). We also list the number of configuration options we left as concrete. Our decision to leave some options concrete was primarily driven by two criteria: whether the option was likely to expose meaningful behaviors, and our desire to limit total analysis effort.

This approach allowed us to run Otter numerous times on each program, to explore different scenarios, and to experiment with different kinds of analysis techniques. We used default values for the concrete configuration options, except the one used to force single-process mode in vsftpd. Grep includes a three-valued string option to control which functions execute the search; for simplicity, we introduced a three-valued integer configuration option and set the string based on this value.

4.6 Emulating the Environment

This study was carried out prior to the development of Otter's environment model (Section 2.12). Thus, this study used much simpler environment emulation, just sufficient for Otter to run on the subject programs. The emulation includes a simple in-memory file system, plus code that emulates the network and concurrency. We discuss two main limitations of the ealier system modeling:

Emulating the network. Since vsftpd and ngIRCd are network programs, they make use to network system calls to communicate with their clients. In our model, many of these system calls, such as inet_ntoa, return *hardcoded* constants (e.g., fixed IP addresses), assuming that these values do not affect code coverage. We emulate a socket using two files, one containing data to be sent and another being a buffer for receiving data. Furthermore, ngIRCd uses poll for multiplexing communications with its clients. To emulate poll, each ngIRCd test was written to precisely define the sequence of *events* that ngIRCd will see via poll. A typical test for ngIRCd is shown in Figure 4.3. In this test, ngIRCd first accepts a client, then receives a sequence of

```
1 int client_fd1 = create_socket();
2 int t = 0;
3 event_accept(client_fd1,t++);
4 event_recv(client_fd1,"NICK␣nick1\r\n",t++);
5 event_recv(client_fd1,"USER␣user1␣x␣x␣:user\r\n",t++);
6 event_recv(client_fd1,"JOIN␣#ch\r\n",t++);
7 event_recv(client_fd1,"WHO␣#ch\r\n",t++);
8 event_end(t++);
```

Figure 4.3: An example of ngIRCd test

IRC commands (of the form "<COMMAND> [<parameter1> [<parameter2> ...]]\r\n") from the client, who logs on as nick1 (NICK) with some user details (USER), followed by a request to JOIN channel #ch and list the people in the channel (WHO). For example, by calling event_recv(client_fd1,"NICK␣nick1\r\n",t), ngIRCd receives the message "NICK nick1\r\n" from client_fd1 in its t-th call to poll.

Emulating concurrency. Otter does not handle multiple processes. However, multiple processes are used in vsftpd's standalone mode and in ngIRCd, To work around this, for vsftpd, in which fork() spawns a subprocess that handles client commands, we interpret fork() as driving the program to that subprocess. (The parent process would simply cycle around a loop and spawn another subprocess, so we ignore it.) For ngIRCd, where the child process parses an IP address and passes the result to the parent, we treat fork() as a branching point—we run both subprocesses, but we ignore the child process's output, instead supplying the input expected by the parent process as part of the test case.

4.7 Data and Analysis

We ran our test suites in Otter, with symbolic configuration options as discussed above. We then performed substantial analysis on the results to explore the configuration space of each subject program. To do this we used the Skoll system, developed and housed at UMD [44]. Skoll allows users to define configurable QA tasks and run them across large virtual computing grids. For this work we used around 40 client machines. The final results reported in this section required about two weeks of elapsed time.

Table 4.3 summarizes Otter's runs. The first group of rows shows the total coverage achieved by the test suites, i.e., the maximum coverage achievable for these test suites considering all possible configurations, except those options and values we left concrete. We manually inspected the uncovered lines and found that approximately another 10% of vsftpd and ngIRCd and 2% of grep comprises code for handling low-level errors. Also, another 11% of vsftpd (in addition to the three files we removed) is unreachable in one-process mode. If we adjust for the error handling and unreachable code, our test suites' line coverage exceeds 80% for all subject programs. Covering the remaining code would in many cases have required adding new mocked libraries, adding further symbolic configuration options, etc. Overall, however, based on our analysis of these systems, we believe that the test cases are reasonably comprehensive and are sufficient to expose much of the configurable behavior of the subject programs.

The next group of rows shows the number of configuration options that appear

	vsftpd	ngIRCd	grep
Coverage			
Line	62%	73%	75%
Block	63%	66%	63%
Edge	56%	61%	58%
Condition	49%	57%	52%
# Examined opts/tot			
Line, Block, Edge	22/30	13/13	17/18
Condition	24/30	13/13	17/18
# Paths			
Line, Block, Edge	30,304	53,205	625,181
Condition	136,320	95,637	764,201
Average # Paths			
Line, Block, Edge	474	375	5,533
Condition	2,130	674	6,763
# Combinations	137,438,953,472	61,834,752	66,650,112

Table 4.3: Summary of symbolic execution.

in at least one path condition (i.e., were constrained in at least one path and thus distinguish different execution paths) versus the total number of options set symbolic. In grep, the one unused option was only "used" when being printed, which did not affect any execution path. In vsftpd, there were 6 unused options total. One case was similar to grep—a configuration option specified a port number, which is ignored by our mock system. Three other options could have been covered with additional tests; the remaining two options cannot be touched without changing the settings of some of the configurations options we left concrete.

Notice that Otter constrains two more options with condition coverage than

under the other metrics. This occurs because, as discussed in Section 4.4, we expand logical operators into sequences of conditionals under condition coverage. For example, under line, block, and edge coverage, the condition **if** (x||1) would be treated as a single branch that Otter would treat as always true. But under condition coverage, the conditional would be expanded, and Otter would see **if** (x) first, causing it to branch on x.

The third group of rows in Table 4.3 shows the number of execution paths explored by Otter and that number averaged across all test cases for each program. While Otter found many thousands of paths, recall that these are actually *all* possible paths for these test suites under any settings of the symbolic configuration options. Had we instead naively run each test case under all possible configuration option combinations[1], it would have required a tremendous number of executions (from 2 to 7 orders of magnitude more than the number of paths) for all the subject programs, as shown in the last row in Table 4.3.

Notice also that expanding logical operators under condition coverage can result in many more paths. This effect is most pronounced for vsftpd, which more than quadruples the number of paths because it contains many logical expressions that test multiple configuration options at once. For example, **if** (x||y||z) would yield at most two paths before expansion, but four paths after.

Figure 4.4 plots the number of paths executed by each test case for each program, both with unexpanded logical operators (L/B/E) and expanded (C). The

[1] For each integer option that can take any integer values, we use the minimum number of settings that will lead to distinct program behaviors under the test suite.

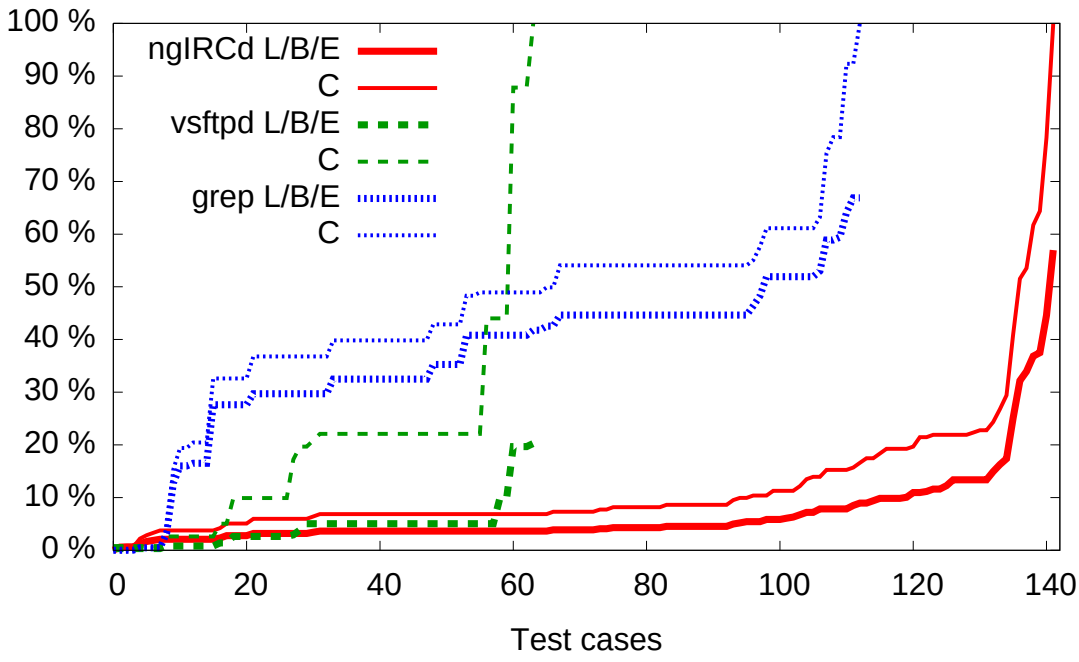


Figure 4.4: Number of paths per test case (L/B/E=line/block/edge, C=condition).

x-axis is sorted from the fewest to the most paths, and the y-axis is the percentage of paths relative to the highest number of paths seen in any test case for the expanded (C) version of the program.

One interesting feature of Figure 4.4 is that, for vsftpd and grep, the numbers of paths of different test cases appear to cluster into a handful of groups (indicated by the plateaus in the graph). This suggests that within a group, the test cases branch on the configuration options in essentially the same manner (most likely because the programs employ common segments of code to test the configuration options). In ngIRCd, this clustering also appears but is less pronounced.

Finally, recall from Table 4.2 that grep, despite still having many fewer paths than configurations, stands out as having a much larger number of paths than the

other programs. We believe this is due to grep's design. In runs of grep with valid inputs, most of grep's code is executed. Therefore many of its configuration options will typically be used, resulting in significant branching in Otter. In contrast, many of vsftpd and ngIRCd's options are not necessarily used in every run. This can be seen clearly in Figure 4.4: only a handful of vsftpd and ngIRCd's tests exercise more than 25% of the paths, while only a handful of grep's tests exercise *fewer* than 25%.

4.7.1 Interaction Strength

Next, we used our guaranteed coverage analysis to explore which configuration option interactions (Section 4.3) are actually required to achieve the line, block, edge, and condition coverage reported in Table 4.3. First, we computed $Cov(true)$, which we call *guaranteed 0-way coverage*. These are coverage elements that are guaranteed to be covered for any choice of options. Here when we refer to *t-way coverage*, t is the interaction strength. Then for every possible option setting $\mathsf{x} = \mathsf{v}$, we computed $Cov(\mathsf{x} = \mathsf{v})$. The union of these sets is the *guaranteed 1-way coverage*, and it captures what coverage elements will definitely be covered by 1-way interactions. Next, we computed $Cov(\mathsf{x1} = \mathsf{v1} \wedge \mathsf{x2} = \mathsf{v2})$ for all possible pairs of option settings, which is *guaranteed 2-way coverage*. Similarly, we continue to increase the number of options in the interactions until $Cov(\mathsf{x1} = \mathsf{v1} \wedge \mathsf{x2} = \mathsf{v2} \wedge ...)$ reaches the maximum possible coverage.

For boolean options, the possible settings are clearly 0 and 1. For integer-valued options, we solved the path conditions discovered by Otter to find possible

concrete settings. For example, if the path condition was `x>=0`, then the solver might choose $\mathsf{x} = \mathsf{0}$ as a possible concrete setting. Because there are multiple path conditions, we sometimes found that different concrete settings were generated by the solver for the same options. In these cases we used our judgement and code examination to select appropriate values.

Table 4.4 shows the number of interactions at each interaction strength. The first thing to notice is that the maximum interaction strength is always seven or less. This is significantly lower than the number of options in each program. We also see that the number of interactions is quite small relative to total number of interactions that are theoretically possible. For example, grep has 14 boolean options, which by themselves lead to (14 choose 2) $\times$ 4 = 728 possible 2-way interactions just with those options alone, yet we see at most 45 2-way interactions for grep.

Also notice that there is not much variation across different coverage criteria—they have remarkably similar numbers of interactions. We investigated further, and we found that the majority of interactions are actually identical across all four criteria. This is an encouraging finding, because it indicates that many interactions are insensitive to the particular coverage criterion.

For ngIRCd, there are significantly more interactions at higher strength than for the other subject programs. This is because almost all of ngIRCd's integer options can take on many different values across our test suite, magnifying the number of interactions.

Finally, we can see that the number of interactions peak around $t = 4$ for vsftpd, $t = 4$ or 5 for ngIRCd, and $t = 2$ or 3 for grep. We believe this corresponds

	t=1	t=2	t=3	t=4	t=5	t=6	t=7
vsftpd							
Line	7	4	3	16	5	6	2
Block	7	4	3	16	6	6	2
Edge	9	4	4	27	7	7	2
Condition	9	4	4	32	14	9	2
ngIRCd							
Line	11	17	31	113	144	111	-
Block	15	22	31	118	147	111	-
Edge	17	26	35	118	159	111	-
Condition	17	30	35	124	174	111	-
grep							
Line	13	27	36	7	5	-	-
Block	14	34	37	7	5	-	-
Edge	23	37	45	11	7	-	-
Condition	23	45	49	16	9	2	-

Table 4.4: Number of interactions at each interaction strength.

to the number of *enabling options* in these programs, discussed more in the next subsection.

4.7.2 Guaranteed Coverage

Figure 4.5 presents the interaction data in terms of coverage. The x-axis is the t-way interaction strength and the y-axis is the percentage of the maximum possible coverage. Note that higher-level guaranteed coverage always includes the lower level, e.g., if a line is covered no matter what the settings are (0-way), then it is certainly covered under particular settings (1-way or higher). As it turns out, the

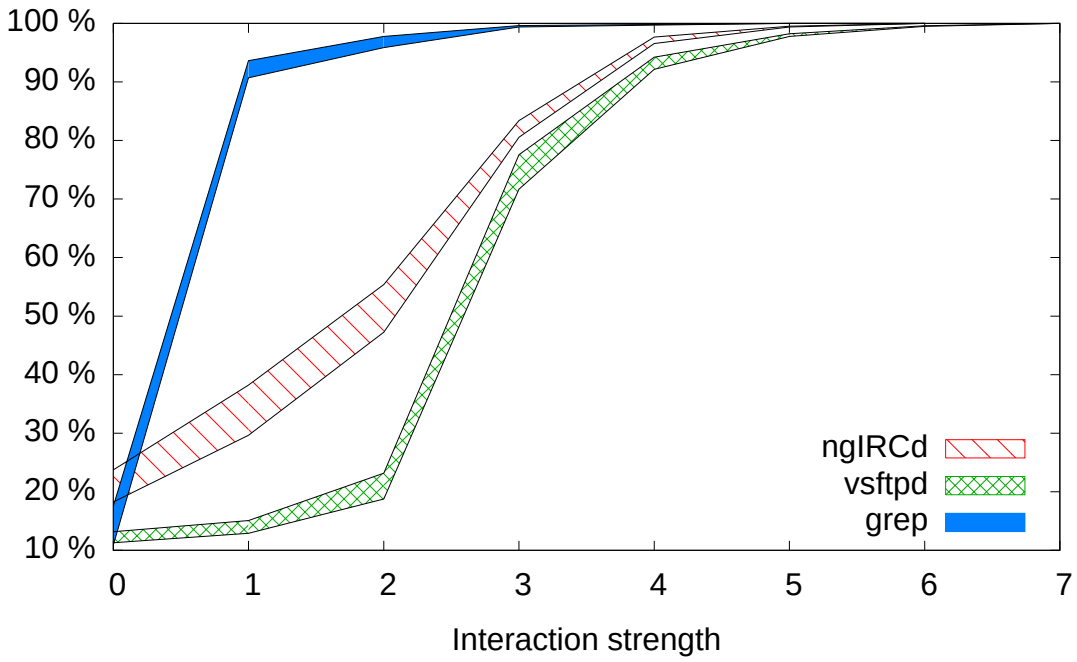


Figure 4.5: Guaranteed coverage versus interaction strength.

trend lines for all four coverage criteria are essentially the same for a given program, and so the plot shows a region enclosing each set of data points. In ngIRCd, the only program with some slightly noticeable variation, line coverage corresponds to the upper boundary of the region, and edge, block, and condition coverage to the lower boundary. This commonality across coverage criteria echoes the same trend we saw in Table 4.4.

One thing to notice in this figure is that the right-most portion of each region adds little to the overall coverage. For these programs and test suites then, high-strength interactions are not needed to cover most of the code. We can also see from this plot that vsftpd gains coverage slowly but then spikes with 3-way interactions, and grep has a similar spike with 1-way interactions. This suggests the presence of *enabling options*, which must be set a certain way for the program to exhibit large

parts of its behavior. For example, for vsftpd (in single-process mode), the enabling options must ensure local logins and SSL support are turned off, and anonymous logins are turned on. For grep, either grep or egrep mode must be enabled to reach most of grep's code; fgrep mode touches little code. ngIRCd also has enabling options that account for the increasing coverage up to interaction strength three, but the effects of these options are less pronounced.

These enabling options also show up in Table 4.4. For example, in that figure we can see that most of vsftpd's interactions are strength $t = 4$ or greater, i.e., they generally involve the three enabling options plus additional options.

4.7.3 Minimal Covering Configuration Sets

Our results so far show that low-strength interactions can cover most of the code. Next, we investigated how interactions can be packed together to form complete configurations, which assign values to all the configuration options. For example, the 1-way interactions `a=0` and `b=0` are consistent and can be packed into the same configuration, but `a=0` and `a=1` are contradictory and must go in different configurations.

We developed a greedy algorithm that packs options together, aiming to find a minimal set of configurations that achieves the same coverage as the full set of runs. We begin with the empty list of configurations. At each step of the algorithm, we pick the interaction that (if we also include the coverage of all subsets of that interaction) guarantees the most as-yet-uncovered lines. Then, we scan through the

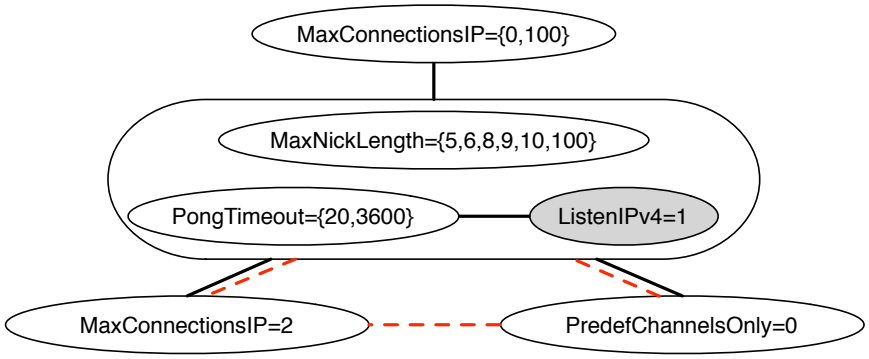
list to find a configuration that is consistent with our pick. We merge the interaction with the first such configuration we find in the list, or append the interaction to the list as a new configuration if it is inconsistent with all existing configurations. This algorithm will always eventually terminate with all lines covered, though it is not guaranteed to find the actual minimum set.

Table 4.5 summarizes the results of our algorithm. The column labeled 1 shows how many lines, blocks, edges, or conditions are covered by the first configuration in the list. Then column n (for $n > 1$) shows the additional coverage achieved by the nth configuration over configurations $1..(n-1)$. Notice that minimal covering sets range in size from 5 to 10, which is much smaller than the number of possible configurations. This suggests that when we abstract in terms of coverage, in fact the configuration space looks more like a union of disjoint interactions (that can be efficiently packed together) rather than a monolithic cross-product.

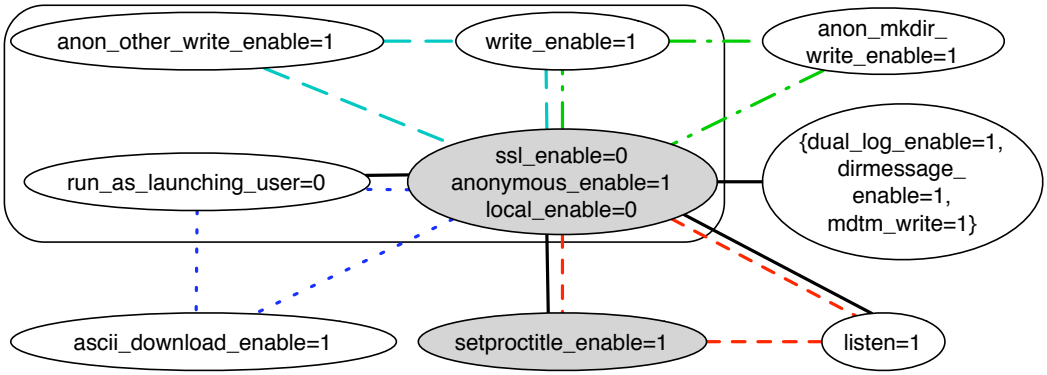
We can also see that each subject program follows the same general trend, with most coverage achieved by just the first configuration in the set. The last several configurations in the set very often add only a single additional coverage element. This last finding hints that not every interaction offers the same level of coverage; we explore this issue in detail next.

4.7.4 Configuration Space Analysis

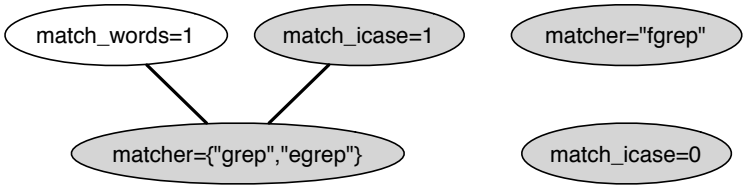
To help visualize interactions and to better understand why the minimal covering sets are so small, we mapped the interactions of each subject program, which



(a) ngIRCd



(b) vsftpd



(c) grep

Figure 4.6: Interactions needed for 95% line coverage. ngIRCd and vsftpd include some approximations.

Config #	1	2	3	4	5	6	7	8	9	10
vsftpd										
Line	2,521	18	8	1	1	-	-	-	-	-
Block	2,853	25	9	1	1	-	-	-	-	-
Edge	2,731	50	17	6	1	1	1	-	-	-
Condition	1,132	71	14	9	2	1	1	1	1	-
ngIRCd										
Line	3,148	30	6	6	1	1	1	-	-	-
Block	4,401	50	8	7	4	1	1	-	-	-
Edge	4,390	62	14	8	6	2	2	2	-	-
Condition	1,881	27	23	6	4	1	1	1	-	-
grep										
Line	2,218	171	34	20	5	5	3	2	2	-
Block	2,838	231	46	28	5	5	3	1	-	-
Edge	3,140	366	51	44	18	9	6	6	4	-
Condition	1,810	231	45	25	11	8	7	6	5	1

Table 4.5: Additional coverage achieved by each configuration in the minimal covering sets.

are shown in Figure 4.6. These graphs show interactions based on line coverage. Because the full set of interactions is too large to display easily, we show only those interactions needed to guarantee 95% of the maximum possible coverage.[2] In these graphs, a node represents one or more option settings; we merged nodes with common neighbors, listing all settings the node represents. 1-way interactions are shaded nodes, 2-way interactions are solid edges, and 3-way interactions are cliques of similarly patterned edges. In Figure 4.6(a), the box denotes a "super node" containing several options, each of which interacts with all three options outside the

[2]The diagrams of the full set of interactions are presented in Appendix B.

box. In Figure 4.6(b), the box instead represents a 4-way interaction. The ngIRCd options are all prefixed with Conf_, and similarly the vsftpd options are prefixed with tunable_. We omitted these prefixes from the graph, however, to save space.

To unclutter the presentation and to highlight interesting interaction patterns, we made some additional simplifications. For ngIRCd, we merged two values for PongTimeout that had similar but not identical neighbor sets, and similarly for MaxNickLength. For vsftpd, we merged the options in the center node of Figure 4.6(b) even though they have slightly different neighbors.

The main feature we see in ngIRCd's graph is the super node in the middle, which contains ngIRCd's enabling options. We can even see their progression: setting ListenIPv4=1 is the first crucial step that lets ngIRCd accept clients, and it forms a 1-way interaction. Next, setting PongTimeout high enough avoids early termination of client connections, and therefore this option forms a 2-way interaction with ListenIPv4=1. The last enabling option, MaxNickLength, forms a 3-way interaction with the previous two. In the full ngIRCd graph, the full set of these enabling options are similarly connected to most of the nodes in the graph.

Next, considering vsftpd's graph, we clearly see that all of the interactions involve the enabling options, which appear in the center, shaded node. There are many interactions involving just one additional option setting, such as the three options in the node at the right middle position. These options control the availability of some features, e.g., dirmessage_enable enables the display of certain messages. Moreover, notice that we can combine all the settings in the nodes of Figure 4.6(b) into one configuration. This helps illustrate why the minimal covering set of configurations

for vsftpd is so small, and why the initial configuration is able to cover so much: one configuration can enable a range of features (writing files, logging, etc.) all at once.

For vsftpd, the full graph of interactions is very much like the image shown here, with a few additional, higher-strength interactions that include the three enabling options, plus a few low-strength interactions, including the other settings for the enabling options, which each guarantee a few additional lines.

Finally, in grep's graph, notice how few configuration options contributed to 95% of the coverage. These high-coverage interactions of grep have very low interaction strength; there are no interactions with strength higher than two, and four out of the five nodes have 1-way interactions. Also, all values of the matcher option appear in this graph, making this the most important option for grep in terms of coverage. The full configuration space graph of grep contains many more interactions and, interestingly, the important matcher option only takes part in a few interactions in the full graph.

While each program exhibits somewhat different configuration space behavior, we can see that when abstracted in terms of line coverage, many options either do not interact or interact at low strength, and thus we can combine them together into larger configurations. This supports our claim that configuration spaces are considerably smaller than combinatorics might suggest.

4.7.5 Threats to Validity

Like any empirical study, our observations and conclusions are limited by potential threats to validity. For example, in this work we used 3 subject programs. Each is widely used, but small in comparison to some industrial applications. In order to keep our analyses tractable, we focused on sets of configuration options that we determined to be important. The size of these sets was substantial, but did not include every possible configuration option. The program behaviors we studied included four structural coverage criteria for this study. Other program behaviors such as data flows or fault detection might lead to different results. Our test suites taken together have reasonable, but not complete, coverage. Individually the test cases tend to be focused on specific functionality, rather than combining multiple activities in a single test case. In that sense they are more like a typical regression suite than a customer acceptance suite. We intend to address each of these issues in future work.

Chapter 5

Other Related Work

In this chapter, I will talk about related work of Chapters 3 and 4.

5.1 Directed Symbolic Execution

ESD [57] is a symbolic execution tool that also aims to solve the line reachability problem. It uses a *proximity-guided path search* that is similar to our SDSE-*intra* algorithm, and an interprocedural reaching definition analysis to find intermediate goals for directing the search. The published results show that ESD works very well on five Coreutils programs we also analyzed (15s for mkdir, 15s for mkfifo, 20s for mknod, 25s for paste, and 11s for tac). Since ESD is not publicly available, we were unable to include it in our experiment directly, and we found it difficult to replicate from its description. One thing we can say for certain is that the interprocedural reaching definition analysis in ESD is clearly critical, as our implementation of SDSE-intra by itself performed quite poorly.

The ESD authors informed us that they did not observe variability in their experiment, which consists of 5 runs per test program [56]. However, we find this somewhat surprising, since ESD employs randomization in its search strategy, and is implemented on top of KLEE whose performance we have found to be highly variable (Table 3.1).

There are major differences between Otter and ESD as well as in the experimental setups that make it hard to compare our results. These include the version of KLEE evaluated (we used the version of KLEE as of April 2011, whereas the ESD paper is based on a pre-release 2008 version of KLEE), symbolic parameters (our analysis uses the same symbolic parameters as in KLEE (except `tac`); the ESD paper did not specify its symbolic parameters used), default search strategy, processor speed, memory, Linux kernel version, whether tests are run in parallel or sequentially, the number of runs per test program, and how random number generators are seeded. These differences may also explain a discrepancy between our evaluations of KLEE: the ESD paper reported that KLEE was not able to find the target bugs within an hour, but in our experiments KLEE was able to find them (note that nearly one-third of the runs for `mkdir` returned within half an hour, which is not reflected by its median).

Several researchers have proposed general, coverage-based symbolic execution search strategies, in addition to the ones discussed in Section 2.13. Burnim and Sen propose several such heuristics, including a distance-based search strategy [10] that directs searches to uncovered branches. It has a different distance calculation, which only considers paths formed by the N nonterminal in Figure 3.2b. Xie et al propose Fitnex, another coverage-based strategy that uses *fitness values* (a measure of how close two predicates are) to guide exploration [55]. It would be interesting future work to compare against these strategies as well; we conjecture that, as these are coverage-based rather than targeted search strategies, they will not perform as well as our approach for targeted search.

Other researchers have proposed different ways to summarize functions to scale symbolic execution. Compositional concolic testing [22, 4] summarizes a function f by a constraint p_f which is a disjunction of constraints, each relating f's input (i.e., parameters) and output (i.e., return value). Then, when generating the next input, any occurence of f can be replaced by p_f in the concolic tester's reasoning, thereby avoiding re-exploring f. The main difference between CCBSE and composition concolic testing is that CCBSE uses partial paths to summarize a *particular* behavior of a function (e.g., how a function fails), whereas composition concolic testing tries to summarize the *entire* function by a (potentially very huge) constraint. Therefore, such method of summarization does not scale to more complex functions.

Researchers have also used model checkers to solve the line reachability problem by specifying the target line as the target state in the model. Much like our work, *directed model checking* [18] focuses on scheduling heuristics to quickly discover the target. Edelkamp et al proposed several heuristics based on minimizing the number of transitions from the current program state to the target state in the model defined by a finite-state automata [19] or Büchi automata [18]. Groce et al suggested using *structural heuristics* such as maximizing code coverage or thread interleavings [27]. Kupferschmid et al borrowed an AI technique based on finding the shortest distance through a *monotonic relaxation* of the model in which states are sets whose successors increase monotonically under set inclusion [32]. In contrast, SDSE prioritizes exploration based on distance in the ICFG, and CCBSE explores backwards from the target.

Directed *incremental* symbolic execution (DiSE [43]), despite its name, solves

a different problem than Chapter 3 solves. The goal of DiSE is, given the old and the new versions of a program, indentify the set of program statements s_i that are affected by the changes, and enumerate *all* execution paths that cover each feasible *permutation* of s_i's exacly once.

5.2 Understanding Configurable Software Systems

Researchers and practitioners have developed several strategies to cope with the problem of testing configurable systems. One popular approach is combinatorial testing [13, 9, 38, 14], which, given an *interaction strength* t, computes a *covering array*, a small set of configurations such that all possible t-way combinations of option settings appear in at least one configuration. The subject program is then tested under each configuration in the covering array, which will have very few configurations compared to the full configuration space of the program.

Several studies to date suggest that even low interaction strength (2- or 3-way) covering array testing can yield good line coverage while higher strengths may be needed for edge or path coverage or fault detection [9, 17, 31]. However, as far as we are aware, all of these studies have taken a black box approach to understanding covering array performance. Thus it is unclear exactly how well and why covering arrays work. On the one hand, a t-way covering array contains all possible t-way interactions, but not all combinations of options may be needed for a given program or test suite. On the other hand, a t-way covering array must contain many combinations of more than t options, making it difficult to tell whether t-way

interactions, or larger ones, are responsible for a given covering array's coverage. Our work attempts to better understand what specific configuration space characteristics control system behavior.

Chapter 6

Future Work

In this chapter, I will briefly sketch some interesting research ideas that will further improve the usefulness of previously discussed symbolic execution techniques.

6.1 Generalization of CCBSE to Finer Program Units

By design, CCBSE generates paths that begin at function entries, and these paths are used to shortcut subsequent searches that hit the corresponding function entries. While in Section 3.3 we saw that shortcutting helps improving runtime (e.g., Mix-CCBSE(OtterKLEE) runs faster than OtterKLEE and CCBSE(RP) on mkdir, mkfifo, mknod and seq), we believe that CCBSE can be even more helpful, if we generalize CCBSE to generate paths that begin at *any* program points. We chose to focus on function entries, because ideally functions have well-defined input (parameters) and output (return value), and naturally become units for compositional analysis like CCBSE. However, for certain reasons, such as poor design of code, a function may be further decomposed into many *logical* functions—blocks of code with clear boundaries. We anticipate that CCBSE (and Mix-CCBSE) will work better on these logical functions, because these functions are smaller and likely to induce shorter (and therefore simplier) paths, thus CCBSE will impose smaller overhead. On the other hand, decomposing a function into logical ones will create

longer call chains, and therefore it will take longer for CCBSE to go back to main or meet the forward search for Mix-CCBSE. Thus it is an interesting question of how to balance these two factors. Furthermore, it is a tricky problem to decide the program points to split a function (candidates include the boundaries between outermost loops or conditionals).

6.2 Better Mix-CCBSE merging algorithm

We observe a weakness of Mix-CCBSE: in order for paths found by CCBSE to be utilized by the forward search, the paths must be found *before* the forward search reaches the function calls corresponding to those paths. For example, consider a program where main calls f that contains the target. Mix-CCBSE works by simultaneously running forward search on main and CCBSE on f. If CCBSE finds a path p from f's entry to the target early enough, then once the forward search reaches the call to f, Mix-CCBSE will try to follow p instead of executing f as usual, thereby having the benefit of shortcutting. However, if p is not discovered before the forward search reaches f, the function f will be executed as usual, and p will be completely ignored for the rest of the execution.

To tackle this issue, we propose a better merging algorithm for Mix-CCBSE. Continuing with the previous example, suppose the forward search is running f for a while and p is just discovered. Our proposed algorithm will look for paths in the forward search that *overlap* with p. We say a path q overlaps another path p if there exists a suffix q' of q, such that q' is a prefix of p. Suppose p' is the suffix of p by

removing q' from p. Then if such path q does exists in the forward search, we can shortcut the execution on q by trying to follow p', thereby making p partially useful.

A challenge is to make this algorithm efficient. However, we speculate the new algorithm will greatly improve the effectiveness of Mix-CCBSE, and therefore we think this idea is worth pursuing.

6.3 Sequential Line Reachability Problem

In Section 3.2 we generalize DSE to multi-target DSE, which finds inputs that drive the execution to multiple targets. We also see another generalization of DSE: given a *sequence* of line targets, extend DSE to find an input that drive the execution to the targets *in the given order*. This generalization arises naturally in reproducing execution that follows a given stack trace (e.g., from an error report). In fact, this is the actual problem that ESD [57] tried to solve; instead, ESD reduced the problem to single-target line reachability, by ignoring intermediate function calls in the stack trace. The consequence is ESD may find a *different* bug than the one that yielded the provided stack trace.

We may solve this problem naively by searching for the first target, and once it is reached we search for the next target, and so on. We anticipate that this strategy will not work well in practice, since each iteration will eliminate a large fraction of feasible answers, leaving no feasible answers to the end. Another naive strategy is to keep searching for paths to the *last* targets, and for each path found, verify if it traverses the given sequence of targets in order. This might work well for "common"

stack traces; otherwise, the probability of success is low.

We conjecture that CCBSE will work well on this problem. By generalizing CCBSE to start symbolic execution at any program point (Section 6.1), we can run CCBSE from the second last target and search for the last target, and repeat the process, until we go back to the first target.

Chapter 7

Conclusions

In this dissertation, I discussed about the architecture and implementation of Otter, a symbolic execution framework for C programs. I also demonstrated how Otter can be used to solve the line reachability problem and to assist the study of configurable software systems.

The line reachability problem arises in automated debugging and in triaging static analysis results, among other applications. We introduced two new directed search strategies, SDSE and CCBSE, that use two very different approaches to solve line reachability. We also discussed Mix-CCBSE, a method for combining CCBSE with any forward search strategy, to get the best of both worlds. We implemented these strategies and a range of state-of-the-art forward search strategies (OtterKLEE, OtterSAGE, and random-path) in Otter, and studied their performance on finding 10 bugs from GNU Coreutils and on three synthetic programs. The results indicate that both SDSE and Mix-CCBSE(OtterKLEE) performed very well in some cases, but they did perform badly sometimes, whereas mixing SDSE with random-path achieves the best overall running time across all strategies. We also generalized our solutions to the line reachability problem to consider multiple line targets, and observed good performance from Mix-CCBSEs and batching the combinations of SDSEs and random-path. In summary, our results suggest that

directed symbolic execution is a practical and effective approach to line reachability.

Furthermore, we have presented an initial experiment using symbolic execution to study the interactions among configuration options for three software systems. Keeping existing threats to validity in mind, we drew several conclusions. All of these conclusions are specific to our programs, test suites, and configuration spaces; further work is clearly needed to establish more general trends. First, we found that we could achieve maximum coverage without executing anything near all the possible configurations. Most coverage was accounted for by lower-strength interactions, across all of line, basic block, edge, and condition coverage. Second, if we packed interactions into configurations greedily, it took only five to ten configurations to achieve this maximal coverage. Third, we also found that in fact it only took one configuration to get the vast majority of the maximum coverage. Finally, by mapping the interactions we gained some insight into why the minimal covering sets are so small. We observed that many options either did not interact or interacted at low strength, and it is often possible to combine different interactions together into a single configuration. Taken together, our results strongly suggest our main hypothesis—that in practical systems, configuration spaces are significantly smaller than combinatorics suggest, and they can be understood from the composition of a small number of interactions.

Appendix A

Tables and Graphs for Directed Symbolic Execution

A.1 Beeswarm distribution plots of benchmark results

A.1.1 Grouped by strategy

The following plots are beeswarm distribution plots generated in `R` [46] using the `beeswarm` [47] package. Each set of plots corresponds to a strategy, and each subplot to a benchmark program from our experiment (Section 3.3). Each point corresponds to the time it takes for a single run to complete. The points are plotted vertically along the y axis, which is scaled to the slowest run that did not time out for each strategy across all benchmark programs, and randomly dispersed horizontally to avoid overlap. Runs that timed out are plotted just above the upper limit of y axis.

A.1.2 Overlaid Pure(S), CCBSE(RP), Mix-CCBSE(S)

To compare Mix-CCBSE strategies against its components, each of the following plots overlays three beeswarm distribution plots: Pure(S), which is the standard forward strategy S, CCBSE(RP), and Mix-CCBSE(S).

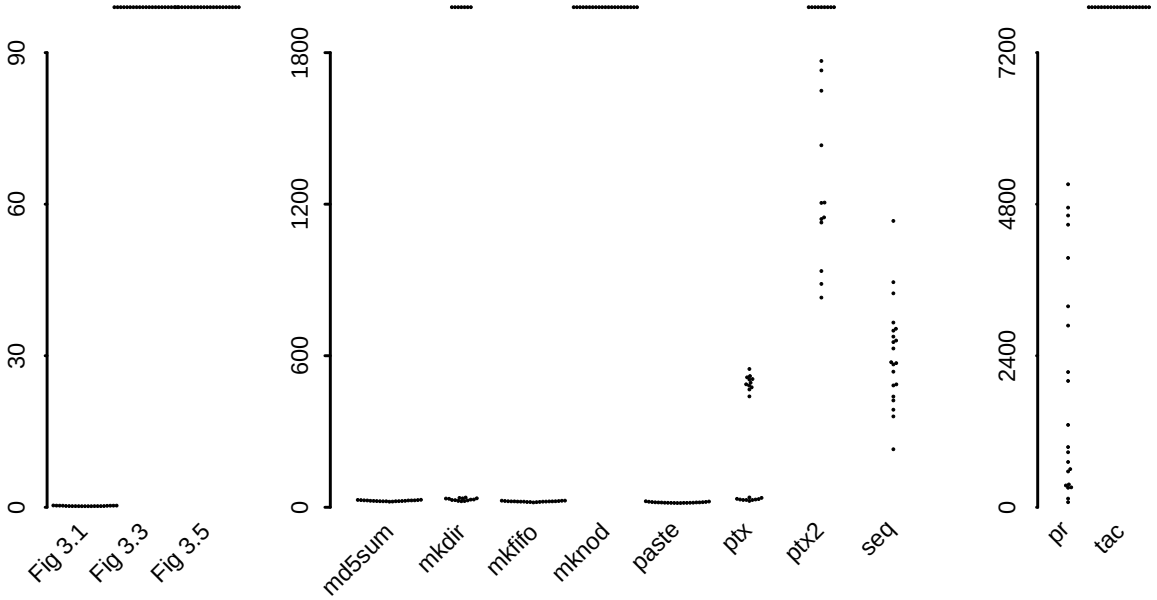


Figure A.1: SDSE

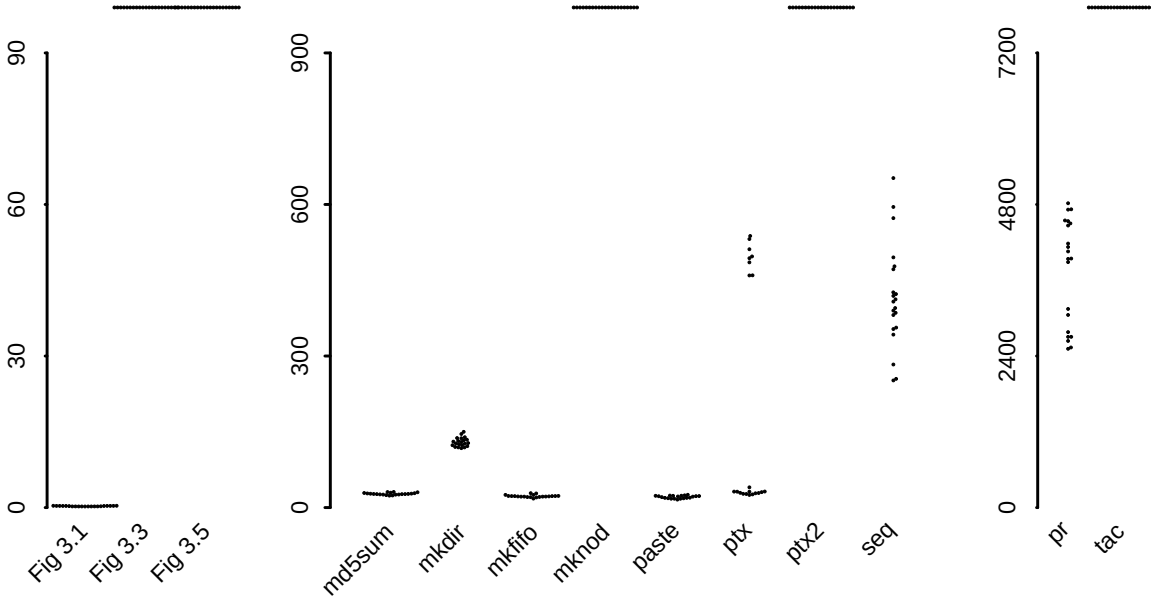


Figure A.2: B(SDSE)

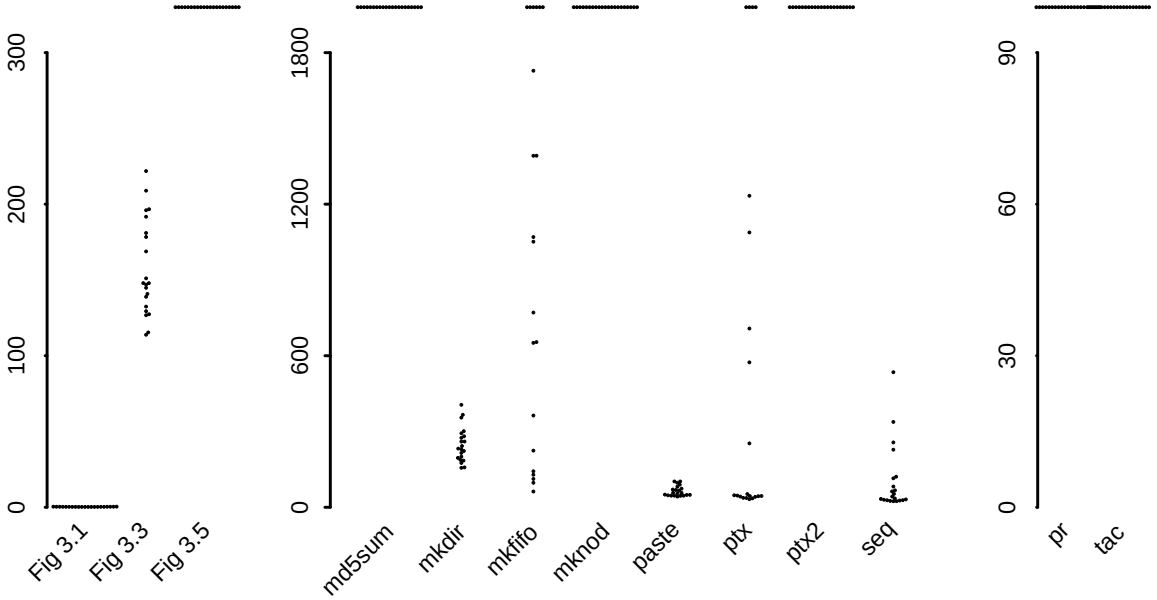


Figure A.3: SDSE-pr

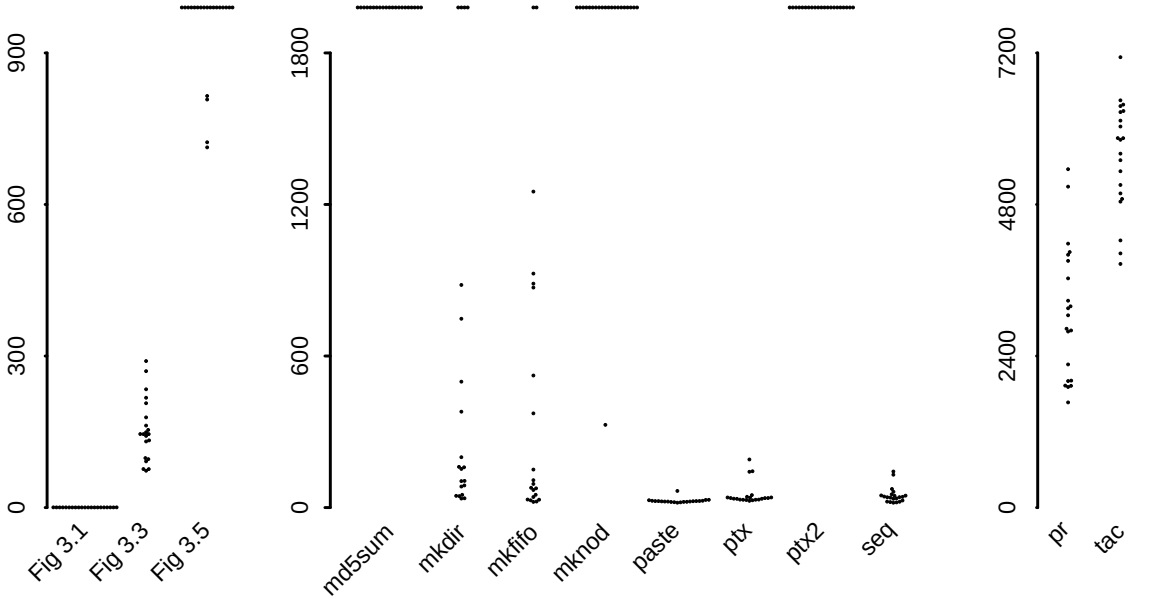


Figure A.4: B(SDSE-pr)

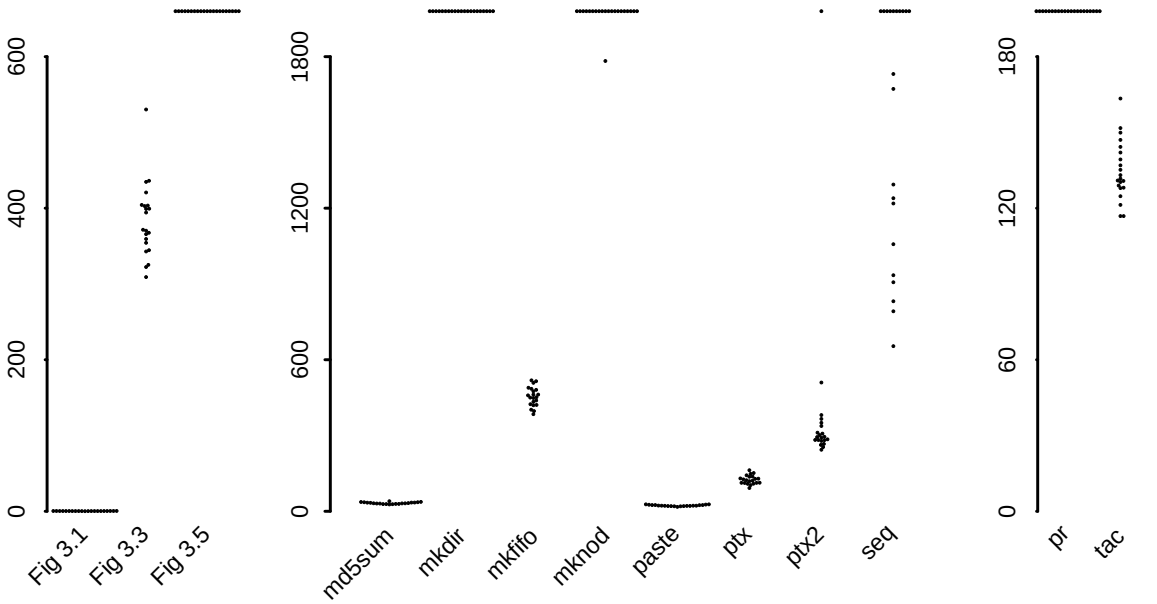


Figure A.5: RR(RP,SDSE)

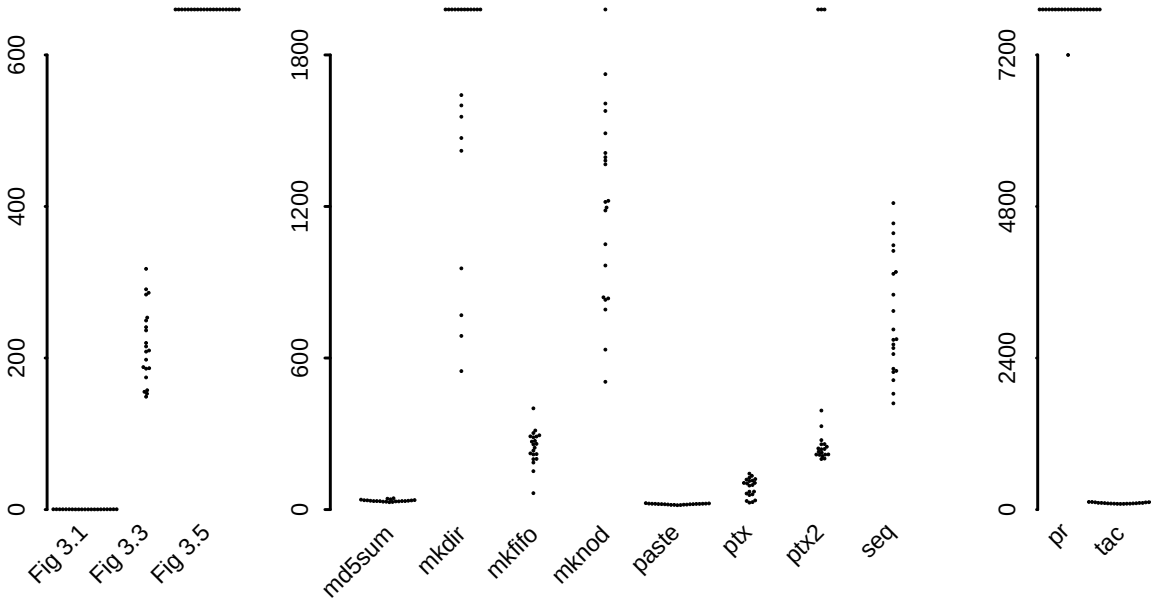


Figure A.6: B(RR(RP,SDSE))

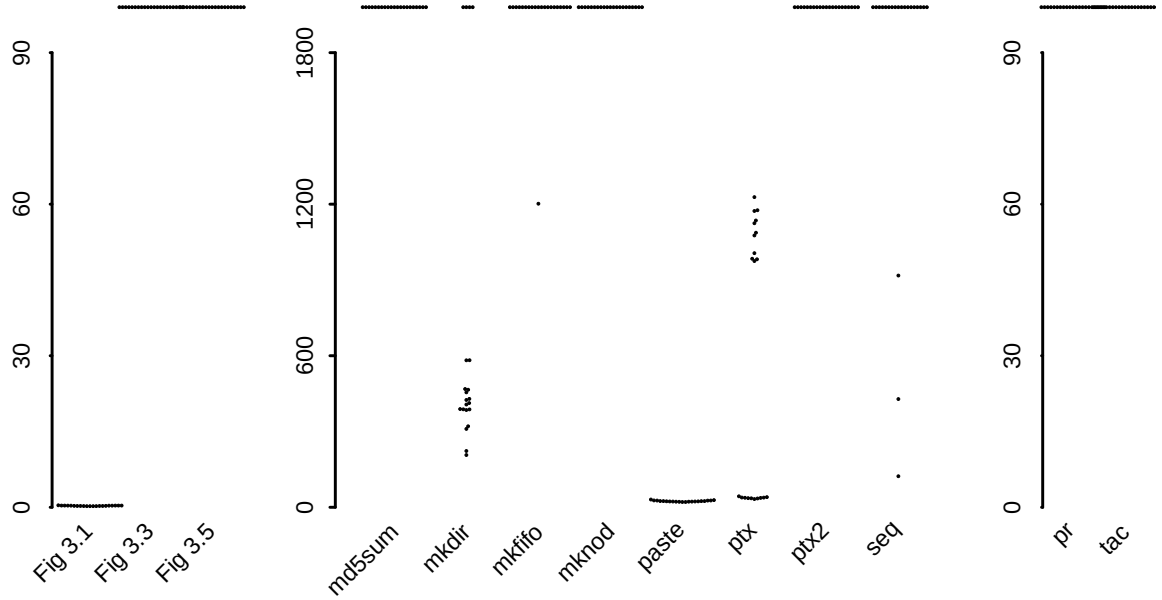


Figure A.7: SDSE-intra

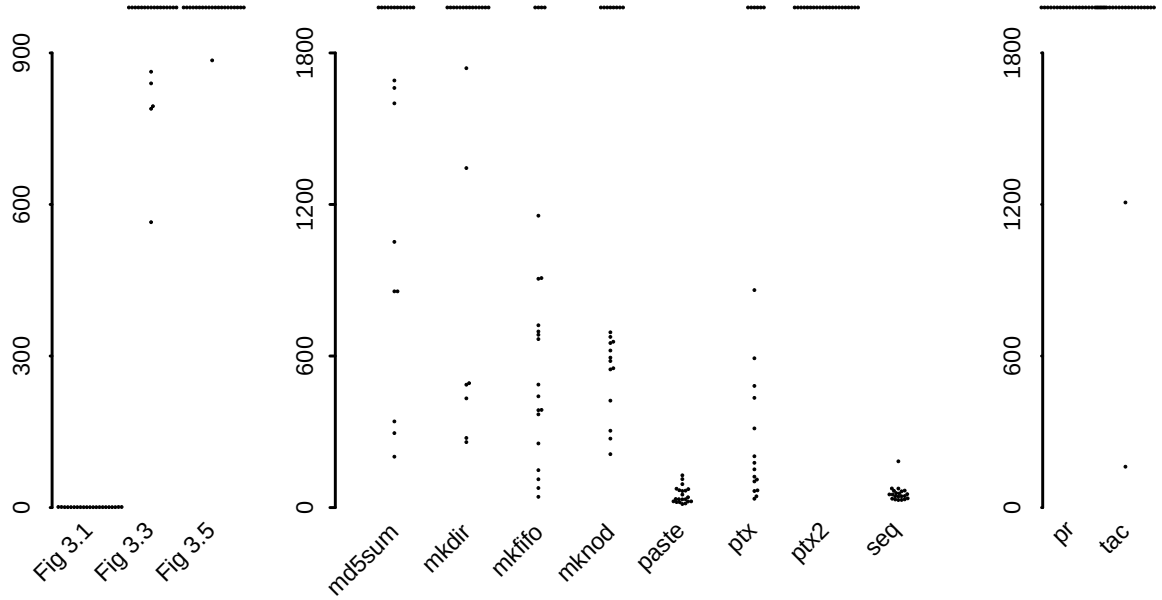


Figure A.8: KLEE

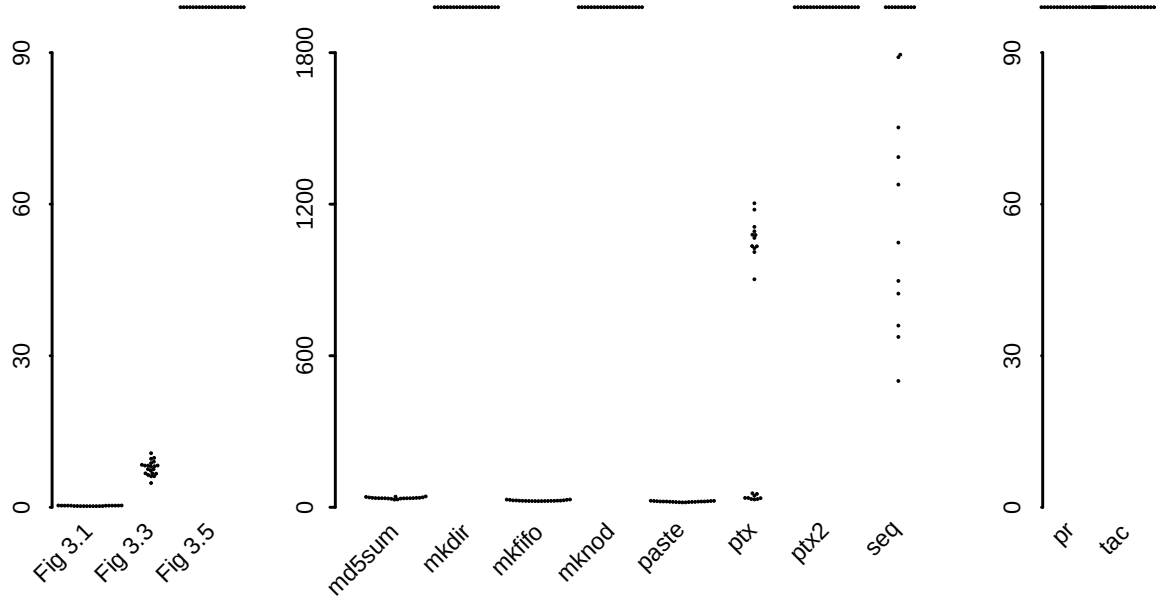


Figure A.9: CCBSE(SDSE)

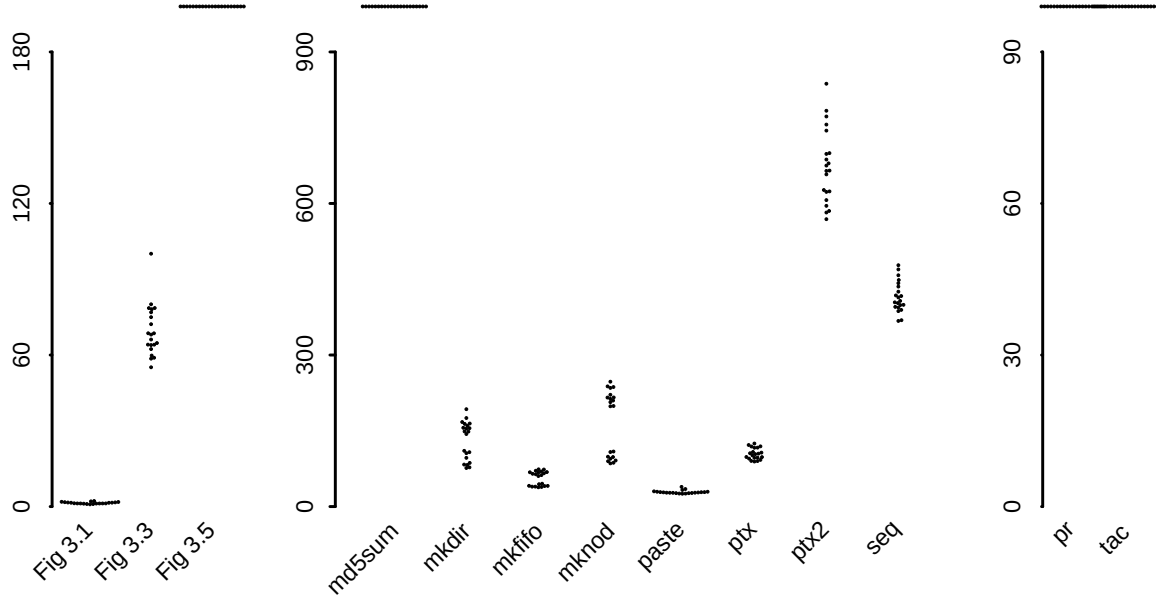


Figure A.10: CCBSE(RP)

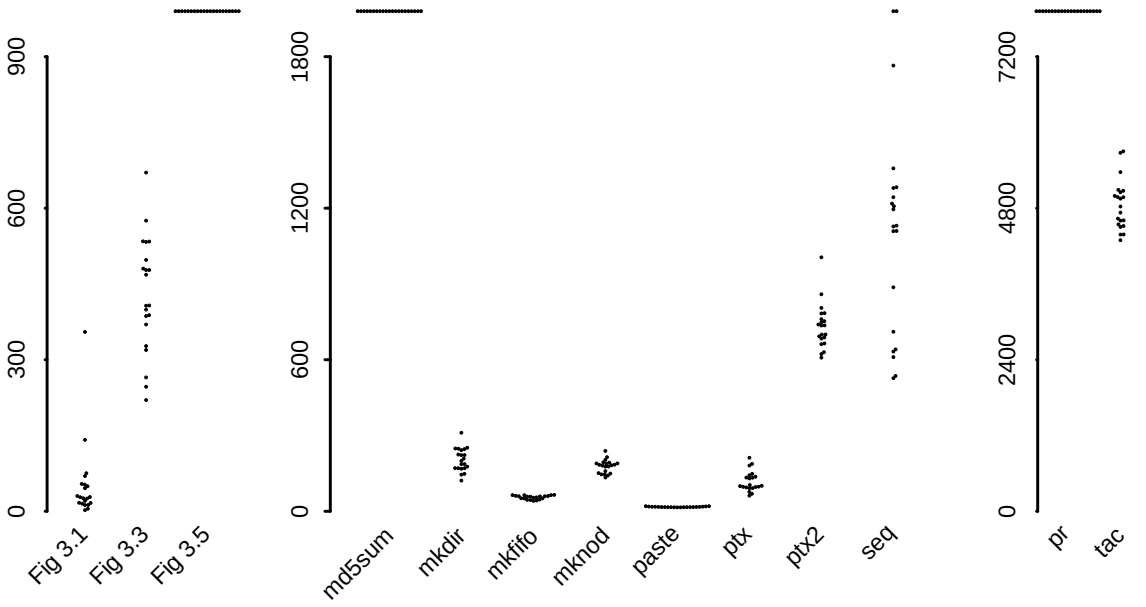


Figure A.11: OtterKLEE

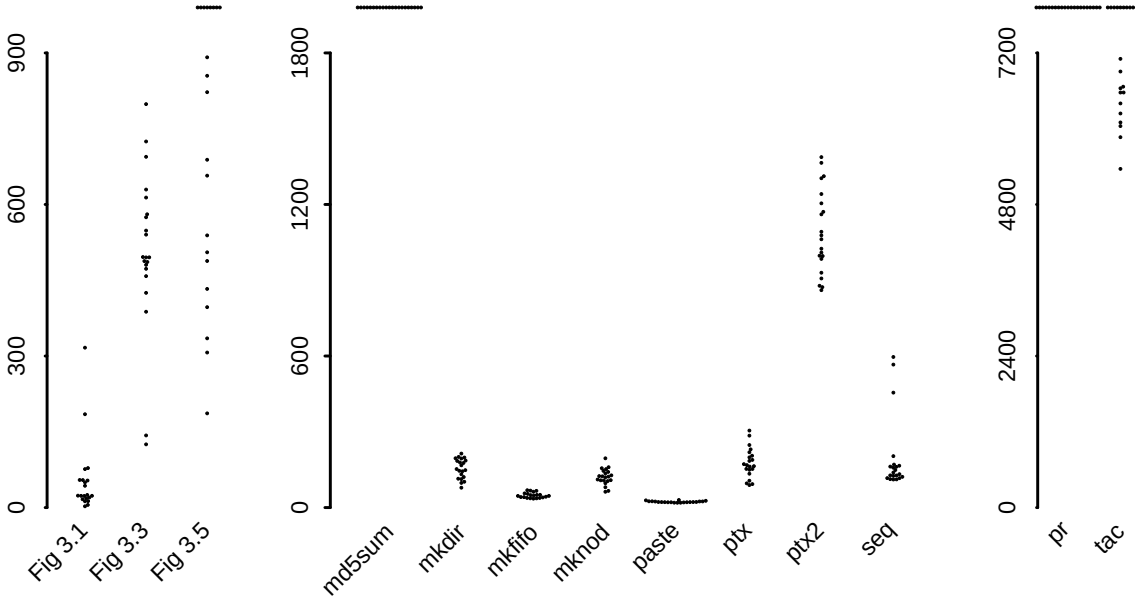


Figure A.12: Mix-CCBSE(OtterKLEE)

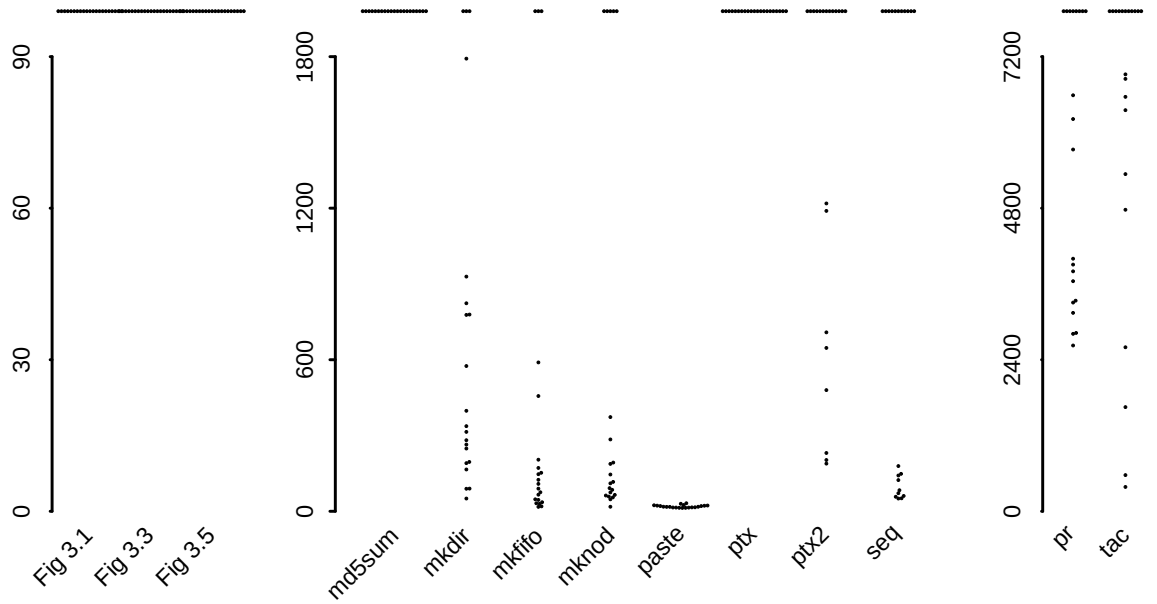


Figure A.13: OtterSAGE

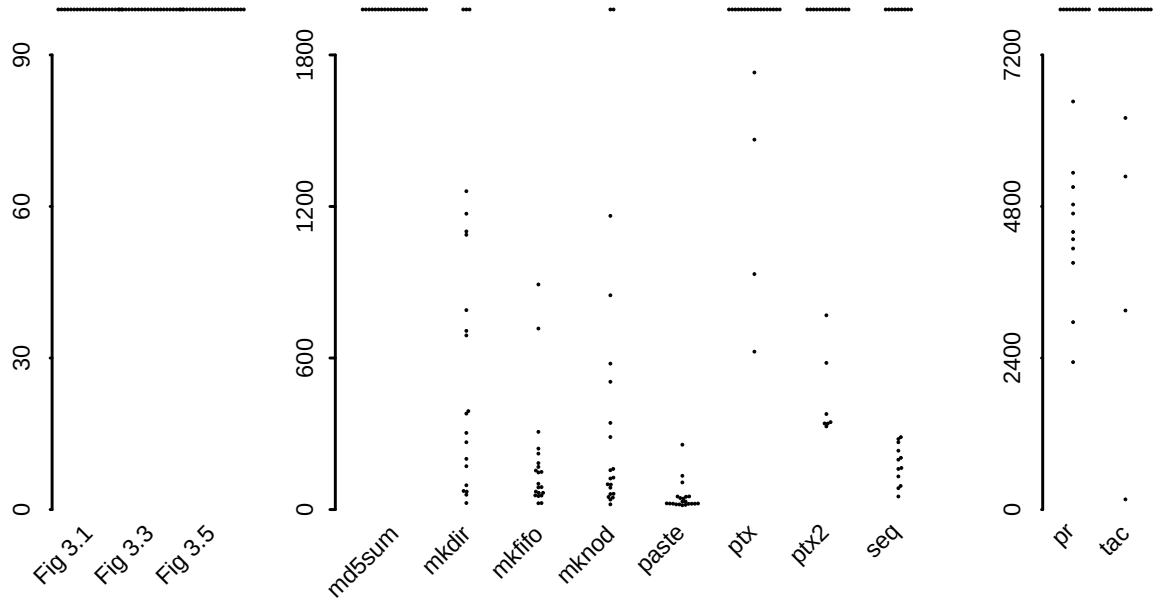


Figure A.14: Mix-CCBSE(OtterSAGE)

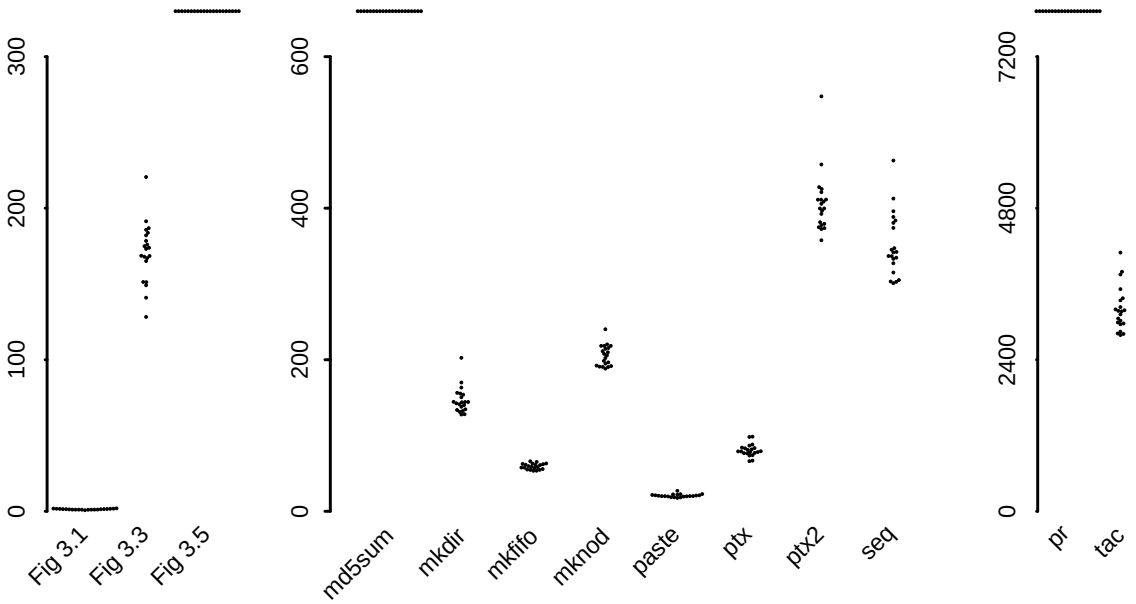


Figure A.15: RP

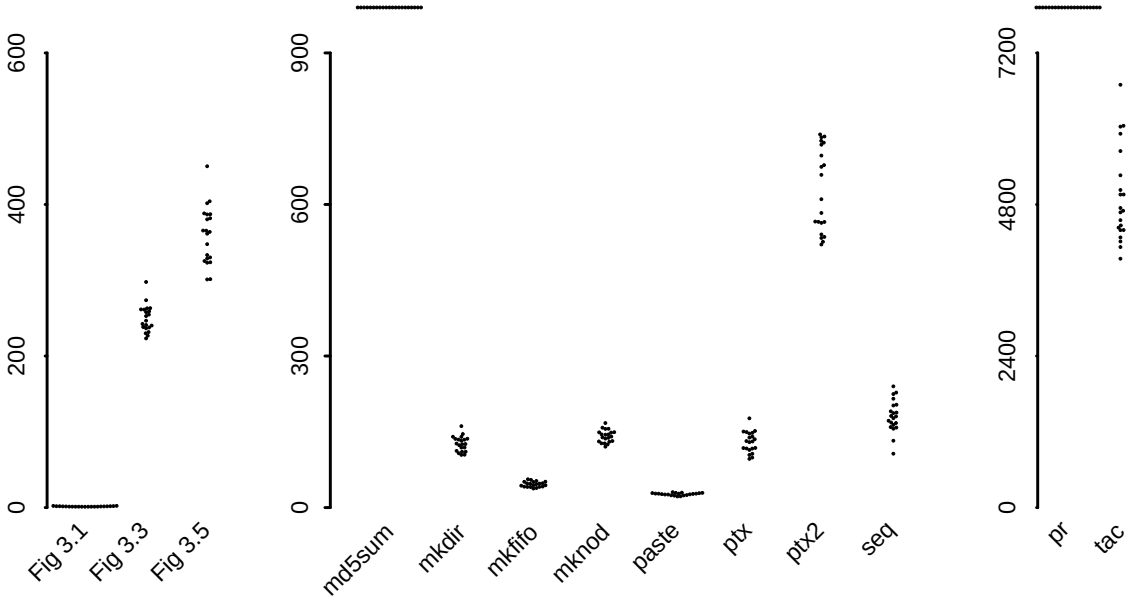


Figure A.16: Mix-CCBSE(RP)

A.1.3 Analysis

Many of the distributions in Appendix A.1.1 are bimodal, which can be seen as two distinct clusters of run times. Since the distributions are observably non-normal, it is inappropriate to summarize our experimental results using mean and standard deviation statistics. Thus, in Tables 3.1 and 3.2, we report the median and SIQR, which are non-parametric (distribution-agnostic) statistics.

Bimodal distribution in CCBSE(RP). CCBSE(RP) is distinctly bimodal for mkdir, mkfifo and mknod. We analyzed these runs and found that, for the faster clusters, CCBSE(RP) found paths from quote to the target line that are also realizable from main. When CCBSE eventually works backwards to main, the search then short-circuits from main through quote to the target line. Thus, these cases demonstrate the advantages of CCBSE.

For the slower clusters, CCBSE(RP) found paths originating from quote that are ultimately not realizable from main. Here, CCBSE(RP) degenerates to pure random-path with overhead: it works backwards to main (which is the overhead), and then finds a different path to the target. Looking at the random-path plot in Appendix A.1.2, we can see that it is indeed the case that the slower cluster in CCBSE(RP) is slightly slower than random-path.

Bimodal distributions due to time outs. The distributions of several other strategy/program test conditions are also bimodal in that runs either finish quickly or time out. KLEE as well as strategies involving OtterSAGE seem to exhibit this

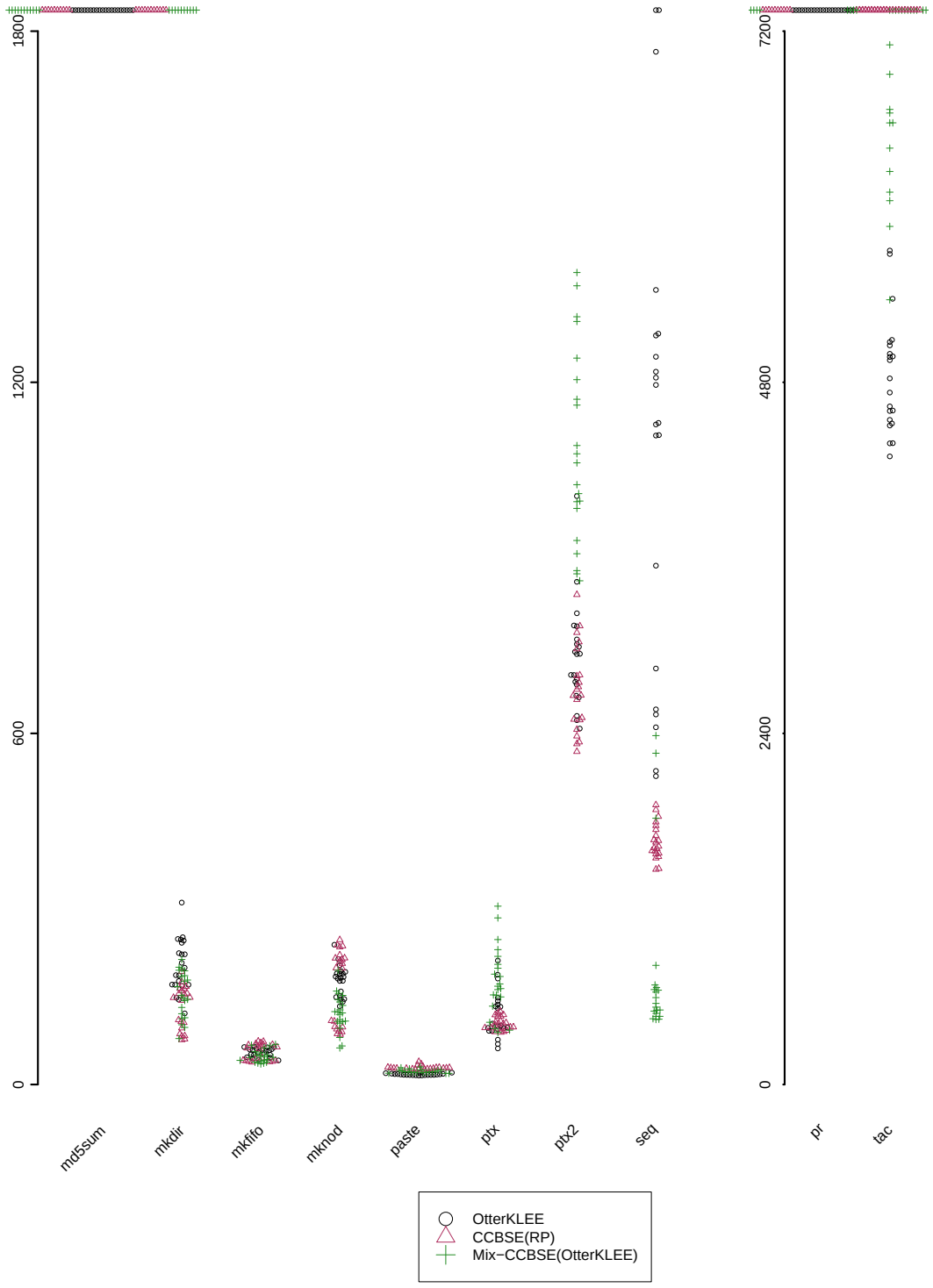


Figure A.17: Overlaying pure OtterKLEE, CCBSE(RP), and Mix-CCBSE(OtterKLEE)

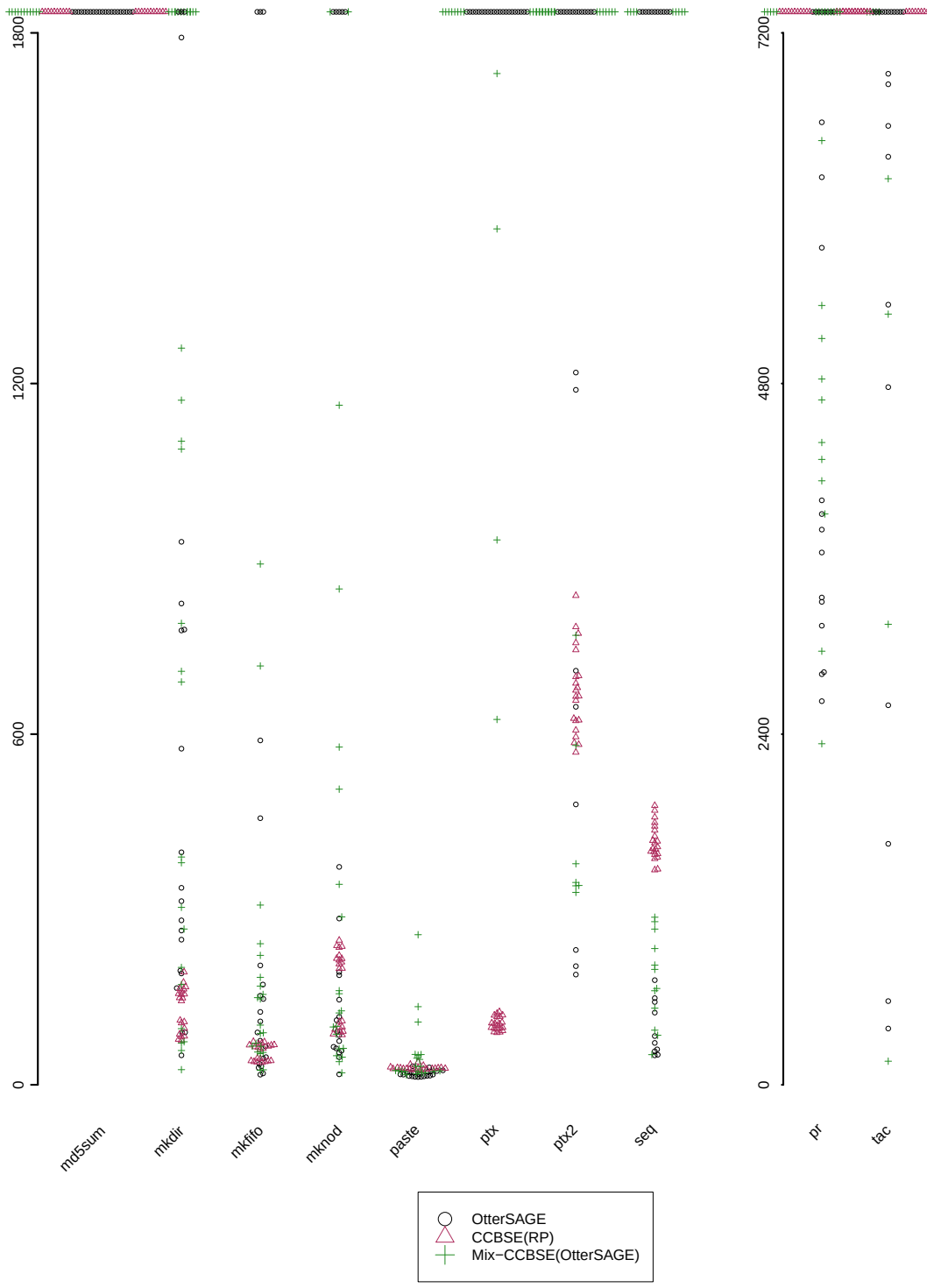


Figure A.18: Overlaying pure OtterSAGE, CCBSE(RP), and Mix-CCBSE(OtterSAGE)

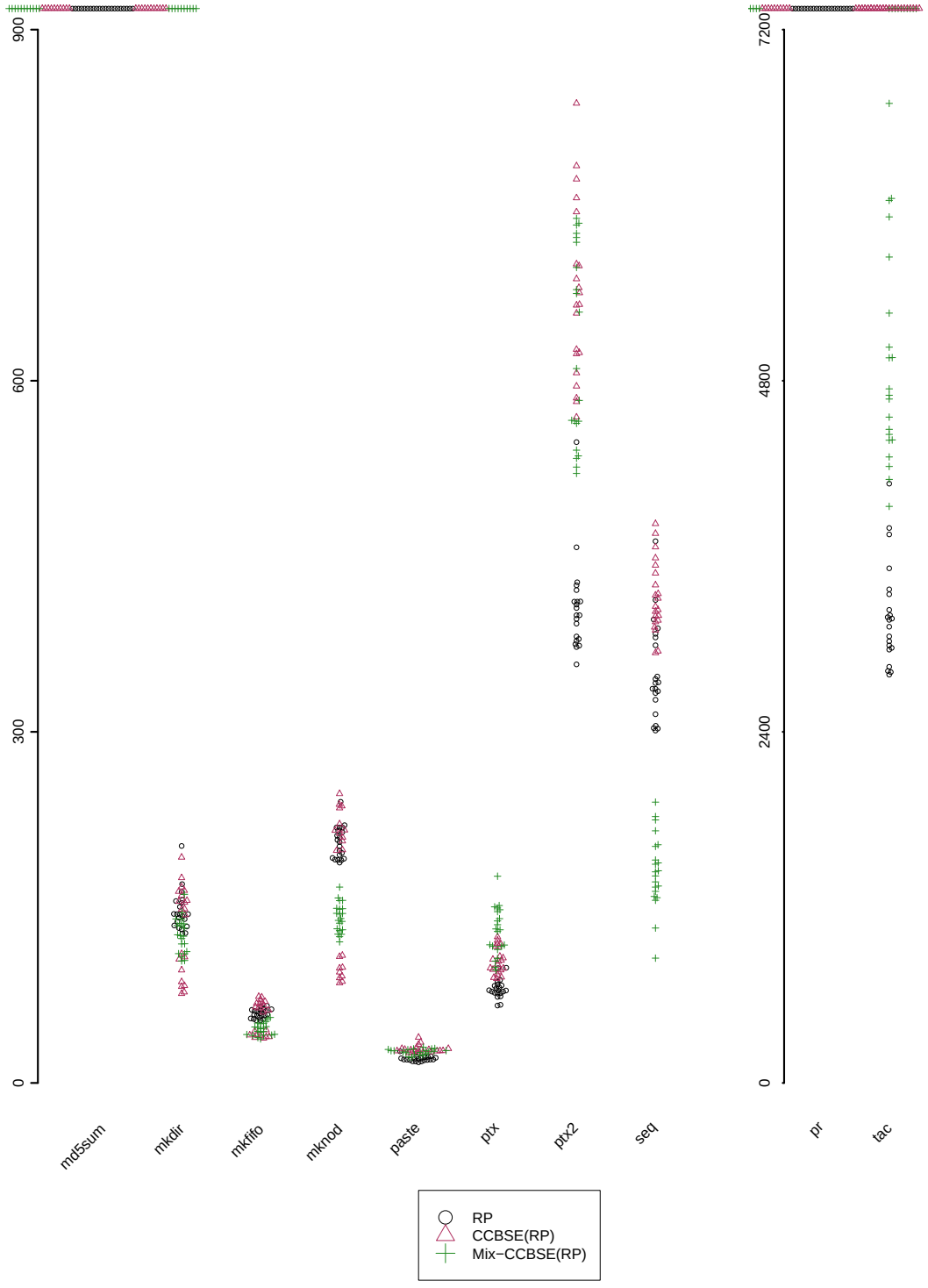


Figure A.19: Overlaying pure RP, CCBSE(RP), and Mix-CCBSE(RP)

issue. For OtterSAGE, we speculate that this is due to its strongly coverage-based heuristic: if a run happens to explore paths that cover many of the same lines as the path to the target, the coverage heuristic may then penalize the path to the target, making it more likely to time out. As a result, the timed-out cluster becomes more distinctly separated from the timely clusters, as seen in the plots.

We observe a much weaker bimodal distribution in OtterKLEE. We believe this is due to OtterKLEE's random-path constituent that helps reducing the penalizing effect. As discussed in Section 3.3.3, OtterKLEE and KLEE are unavoidably different. But in general, randomness in a strategy can lead to exploration that never reaches the target in certain programs, therefore creating two clusters of timely and timed-out runs. This explains the bimodal distribution observed in KLEE.

Mix-CCBSE. In Section 3.1.3, we explained that we mix strategies with CCBSE in order to get the best of both worlds, but it can as well degenerate to being worse than either. The plots in Appendix A.1.2 show some examples of the former.

For OtterKLEE and random-path, Mix-CCBSE (as shown by green crosses) tends to be located towards the middle to the bottom of the distribution for each program; in the case of Mix-CCBSE(OtterKLEE) for `mknod` and `seq`, it is located at the bottom, i.e., Mix-CCBSE(OtterKLEE) performs better than either of its constituents alone.

The analysis for OtterSAGE is less positive: Mix-CCBSE(OtterSAGE) seems to be as bad as OtterSAGE alone. We speculate that this is because OtterSAGE will always run a path to completion, even if the path has reached a point in the

program where the target is no longer reachable, and Mix-CCBSE(OtterSAGE) can no longer take advantage the partial paths found by CCBSE.

A.2 Coverage-over-time Plots in Multi-target Experiment

Figures A.20-A.28 show, for each Coreutils program, the coverage over time for different strategies studied in multi-target DSE (Section 3.2). To prepare a plot for each program, we first compute the median time each strategy uses to cover a target, then we sort the targets according to their discovery times for each strategy. The result is a strictly increasing curve, where the ith point marks the time a strategy takes to cover the ith target. The time axis is shown in log scale.

Notice that curves in the same plot are not strictly comparable, in the sense that different strategies may find different subsets of targets, or find the targets in different orders. Nevertheless, we find these plots very useful for showing coverage *rates* of different strategies, especially if we assume that targets are of equal importance.

We analyze and comment on the trend of coverage of different strategies for each program. Our general observation is that, while undirected strategies (KLEE, SAGE and RP) might cover targets faster in the beginning, certain directed strategies (in particular B(RR(RP,SDSE)), B(RR(RP,SDSE-rr)) and Mix-CCBSE strategies) made gradual progress and achieved higher coverage than undirected strategies in the end. This shows that directed strategies are useful in solving multi-target line reachability problem.

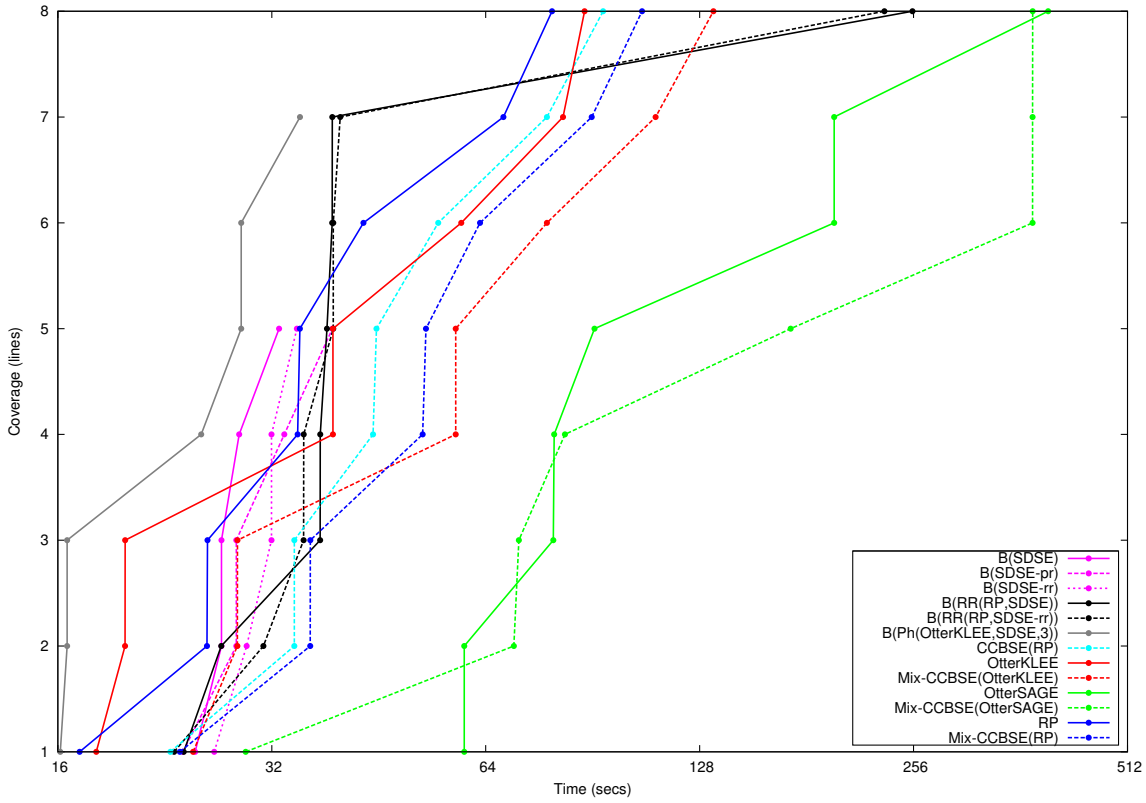


Figure A.20: mkdir (8 targets). Many strategies achieved the full coverage. RP and OtterKLEE reached the full coverage first, while CCBSE(RP), Mix-CCBSE(RP) and Mix-CCBSE(OtterKLEE) achieved the same coverage slightly slower. B(RR(RP,SDSE)), B(RR(RP,SDSE-rr)), OtterSAGE and Mix-CCBSE(OtterSAGE) were able to achieve the same coverage within reasonable time. Notice, however, that B(Ph(OtterKLEE,SDSE,3)) found the first 7 targets much earlier than other strategies (including OtterKLEE).

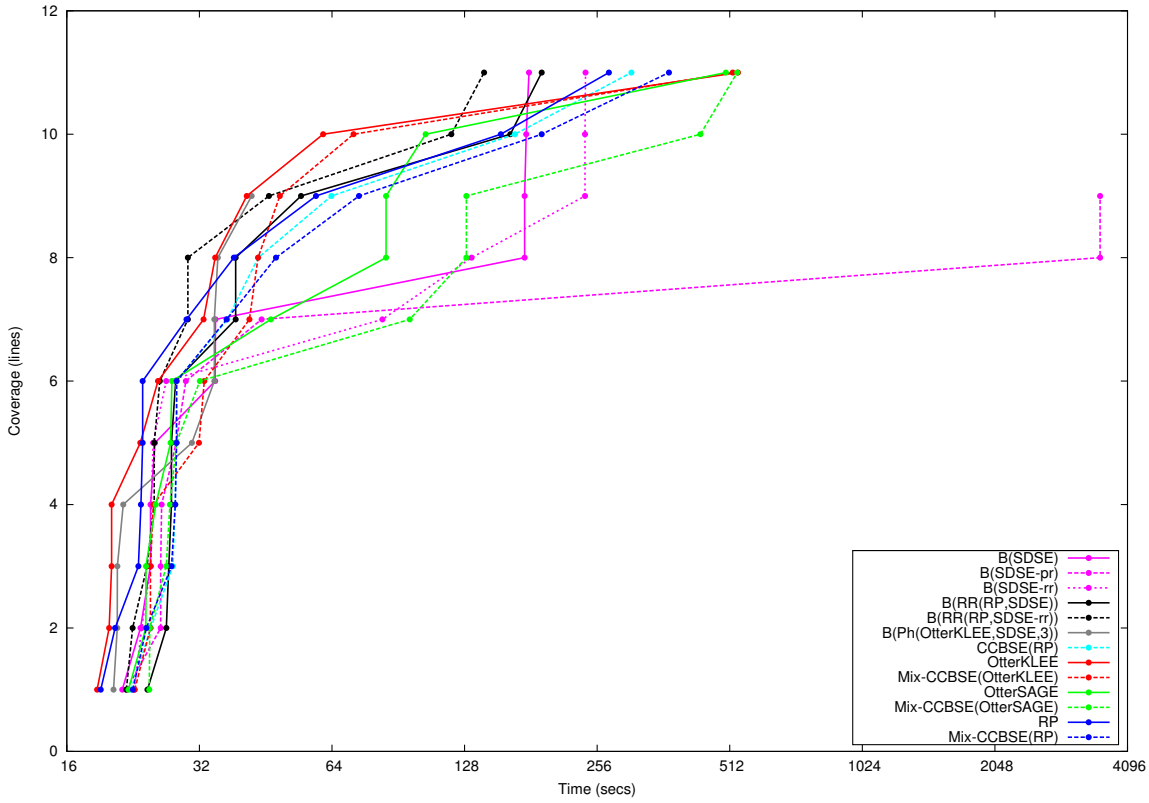


Figure A.21: mkfifo (11 targets). OtterKLEE and RP found the first 6 targets slightly earlier than other strategies. OtterKLEE was able to maintain the pace up to the 10th target, However, B(RR(RP,SDSE-rr)) and later B(RR(RP,SDSE)) and B(SDSE) caught up and reached full coverage earlier than the undirected strategies.

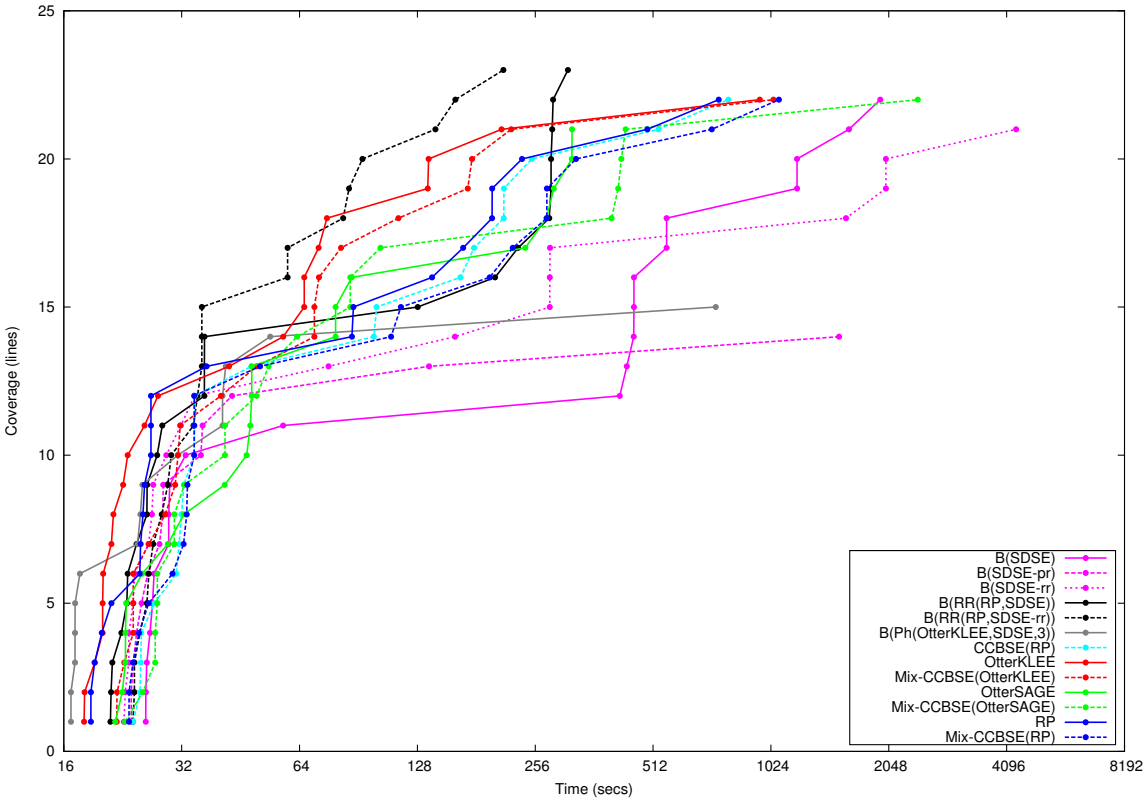


Figure A.22: mknod (23 targets). We observe that B(Ph(OtterKLEE,SDSE,3)), OtterKLEE and RP led in the race in the beginning, but B(RR(RP,SDSE-rr)) gradually increased its coverage, and together with B(RR(RP,SDSE)) they were the only strategies achieving full coverage.

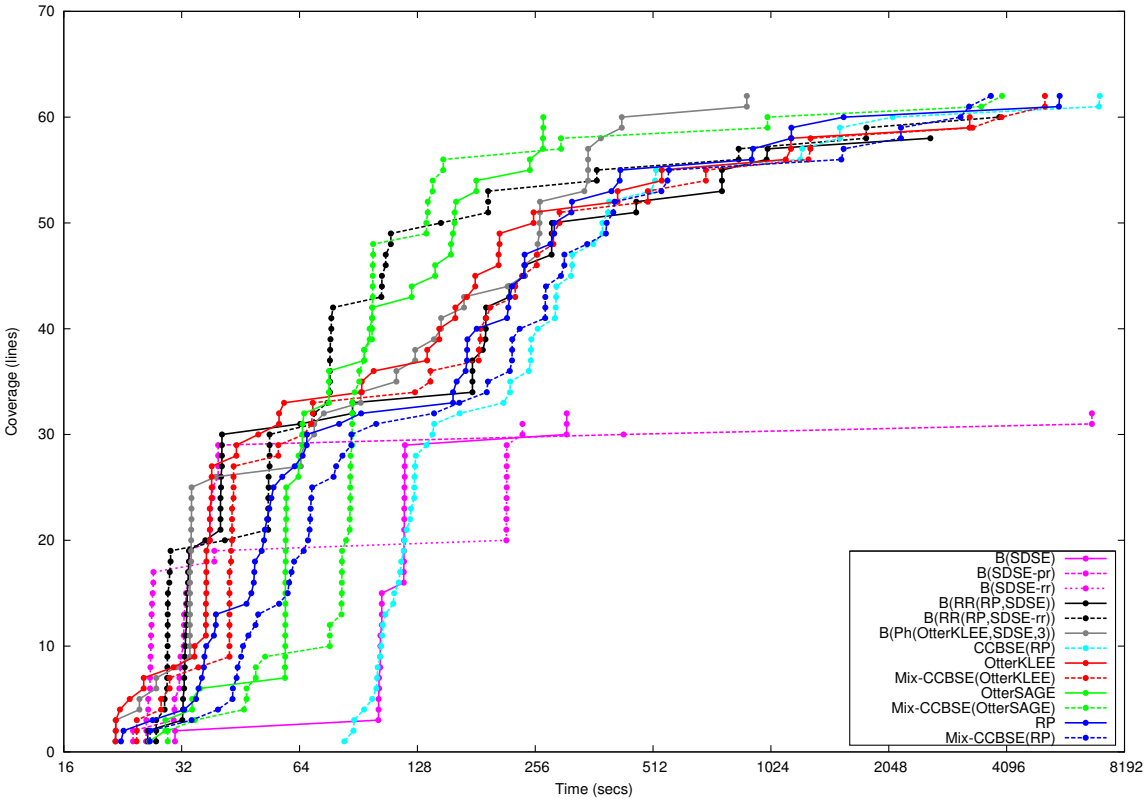


Figure A.23: paste (78 targets). Many strategies achieved the same highest coverage, however B(Ph(OtterKLEE,SDSE,3)) was the fastest strategy that achieved it. On the other hand, pure SDSE strategies performed poorly.

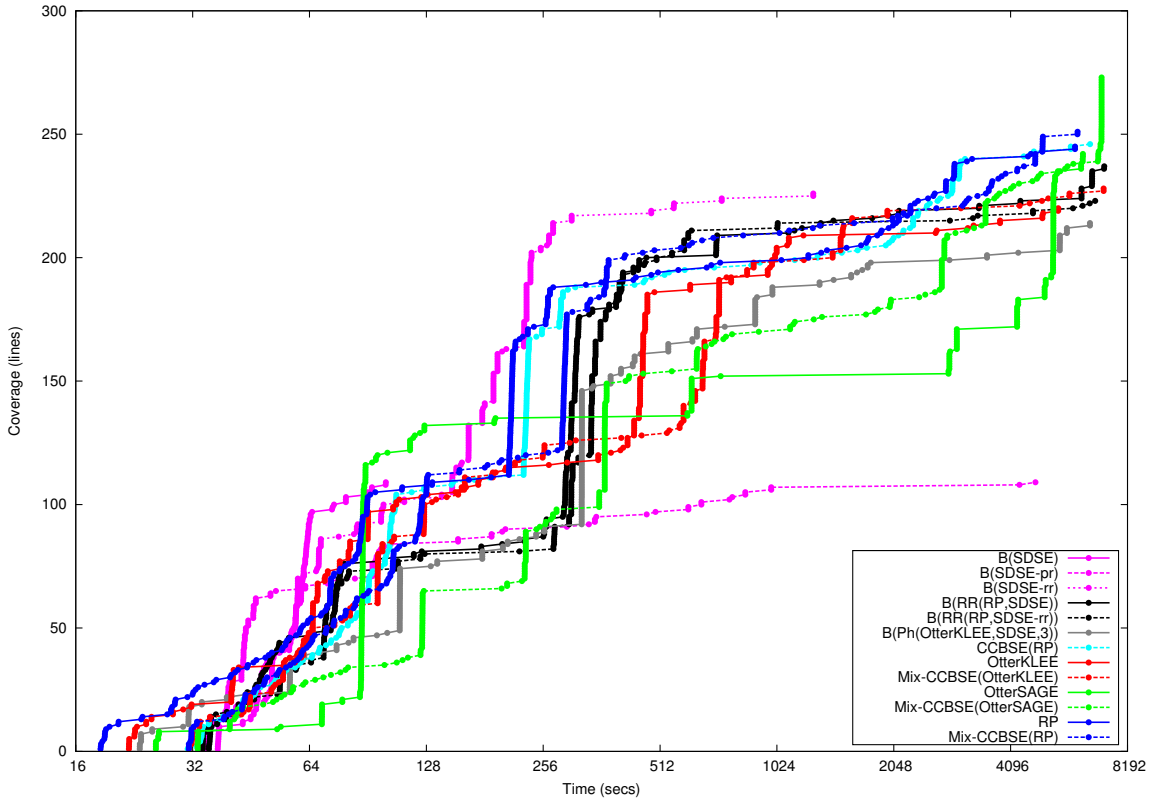


Figure A.24: ptx (517 targets). Mix(OtterSAGE) performed very well here. OtterSAGE and the batched SDSEs covered quickly in the beginning. However, only Mix(OtterSAGE) kept increasing coverage and obtained the highest coverage in the end.

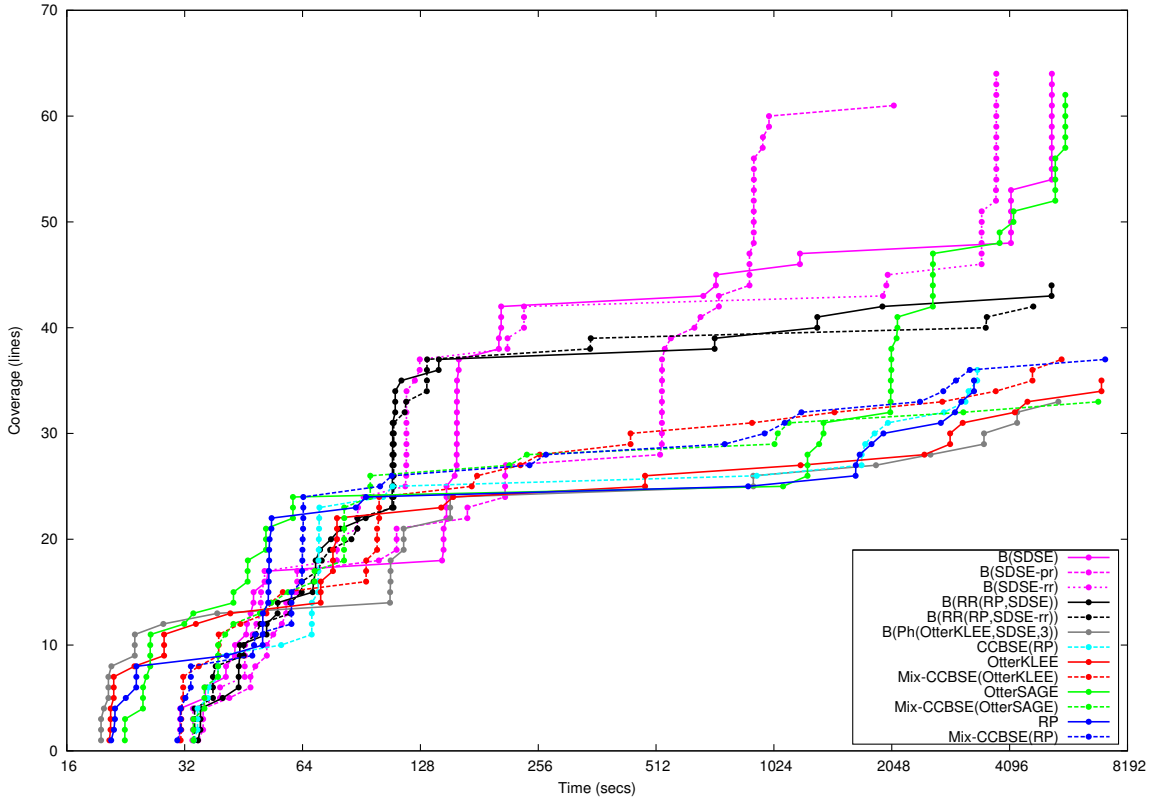


Figure A.25: pr (92 targets). We observe that SDSE strategies dominated others. While they were slow in the beginning, B(SDSE) and B(SDSE-rr) caught up quickly and achieved the highest coverage in the end. B(RR(RP,SDSE)) and B(RR(RP,SDSE-rr)) performed far worse, showing that random-path affects SDSEs and ruins their effectiveness.

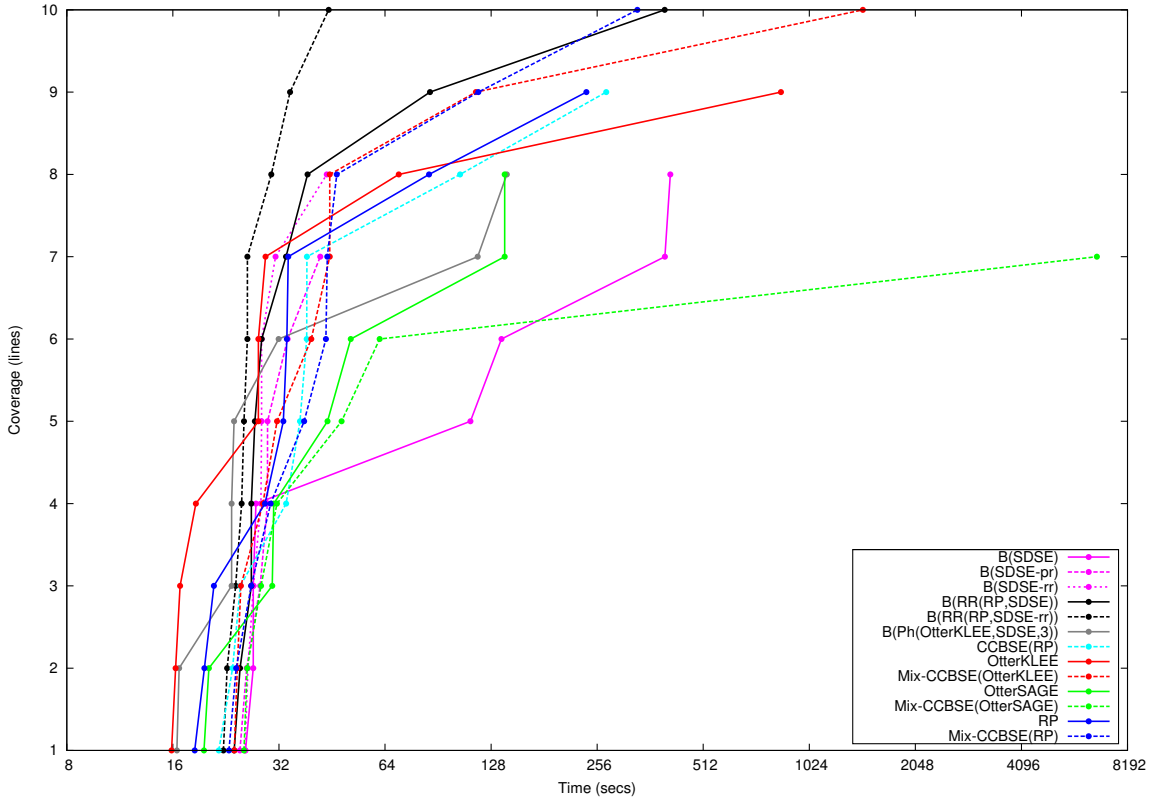


Figure A.26: seq (16 targets). OtterKLEE was leading in the beginning, but B(RR(RP,SDSE-rr))'s caught up and won in the end.

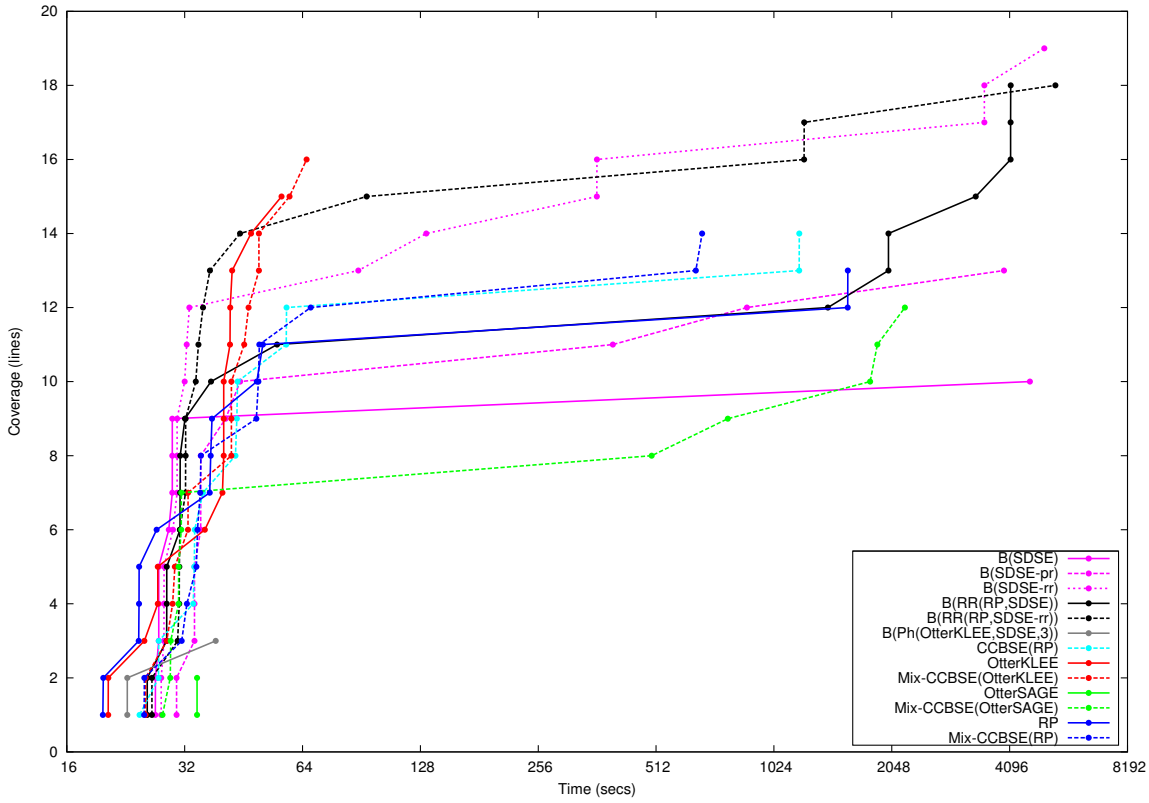


Figure A.27: md5sum (65 targets). RP and OtterKLEE were mostly leading in the beginning, but then B(SDSE-rr), B(RR(RP,SDSE)) and B(RR(RP,SDSE-rr)) achieved more coverage. B(SDSE-rr) got the highest coverage in the end.

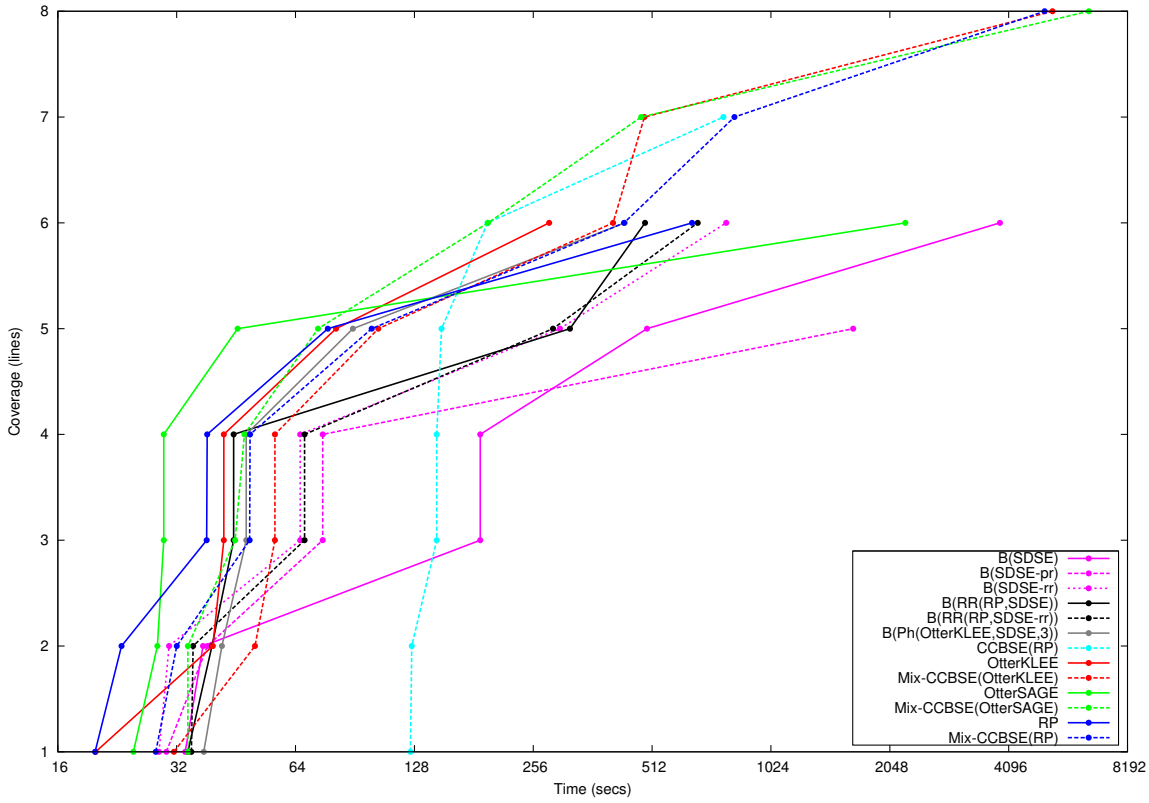


Figure A.28: tac (51 targets). All the strategies did not good very good coverage. But comparing among themselves, Mix-CCBSE strategies performed better, while the SDSE strategies had average performance. We observe that, all the undirected strategies (OtterKLEE, OtterSAGE and random-path) performed better than their Mix-CCBSE versions in the beginning, however these undirected strategies did not make further progress, while the Mix-CCBSE strategies gradually made progress and finally achieved good coverage.

Appendix B

Interactions due to Line Coverage

The figures below depict the entire set of interactions due to line coverage for each of our subject programs: ngIRCd, grep, and vsftpd. In these graphs, a node is shaded if it guarantees coverage on its own, black edges represent interactions involving just two nodes, and interactions involving more than two nodes are cliques of similarly patterned and similarly colored edges. Nodes represent one or more option settings; we merged nodes with common neighbors, listing all settings the node represents. The ngIRCd options are all prefixed with Conf_, and similarly the vsftpd options are prefixed with tunable_; we omit these prefixes to save space.

In each of our programs, there were some settings that were involved in many interactions. In ngIRCd, this is

ListenIPv4=1; in vsftpd, it is a 3-way interaction among

ssl_enable=0, local_enable=0, and anonymous_enable=1; and in grep, it is a 2-way interaction between match_words=0 and match_lines=0. For vsftpd and grep, we grouped this key interaction into a single node. Then, to help keep the graphs legible, we omitted the edges incident on these key nodes for interactions involving more than one other node. Instead, in Figure B.1, interactions involving the key node are marked by thin edges while others are marked by thick edges; Figures B.2 and B.3 have the roles of thick and thin edges reversed.

ngIRCd is depicted differently than grep and vsftpd; many of ngIRCd's option settings have *nearly* identical neighbors as some other settings, so most options are depicted as a single node which contains all of the possible values for that option. When multiple values of an option interact with the same other settings, a single edge is used to represent *all* such interactions, with the set of values for these interactions enclosed together in a subnode of the option's node. For example, the thin black edge connecting the MaxNickLength node to the values 20 and 3600 of PongTimeout represents 10 different 3-way interactions: the interaction among ListenIPv4=1 (indicated by the line being thin), each of the 5 values of MaxNickLength, and each of PongTimeout=20 and PongTimeout=3600. (The colors of the subnodes of MaxNickLength are only to help distinguish the subnodes one from another.)

Two options, UID and ListenIPv4, are not depicted with a single node containing all the values because both options' settings have very few edges in the graph, so this would not have helped keep the graph sparse.

While the graphs are intended to give a rough sense of what options interact and how, they are difficult to decipher, even with our attempts to keep them tidy. Therefore, we also list the interactions themselves in Figures B.4 through B.7.

Finally, in Figure B.8, we list the entire set of options we set symbolic during our tests. For the non-boolean options, some had constraints on what values they could take, either implicitly in the program, or imposed by us (in an attempt to maximize coverage while keeping symbolic evaluation practical); the figure lists their possible values. The remaining options were integer-valued options on which we put

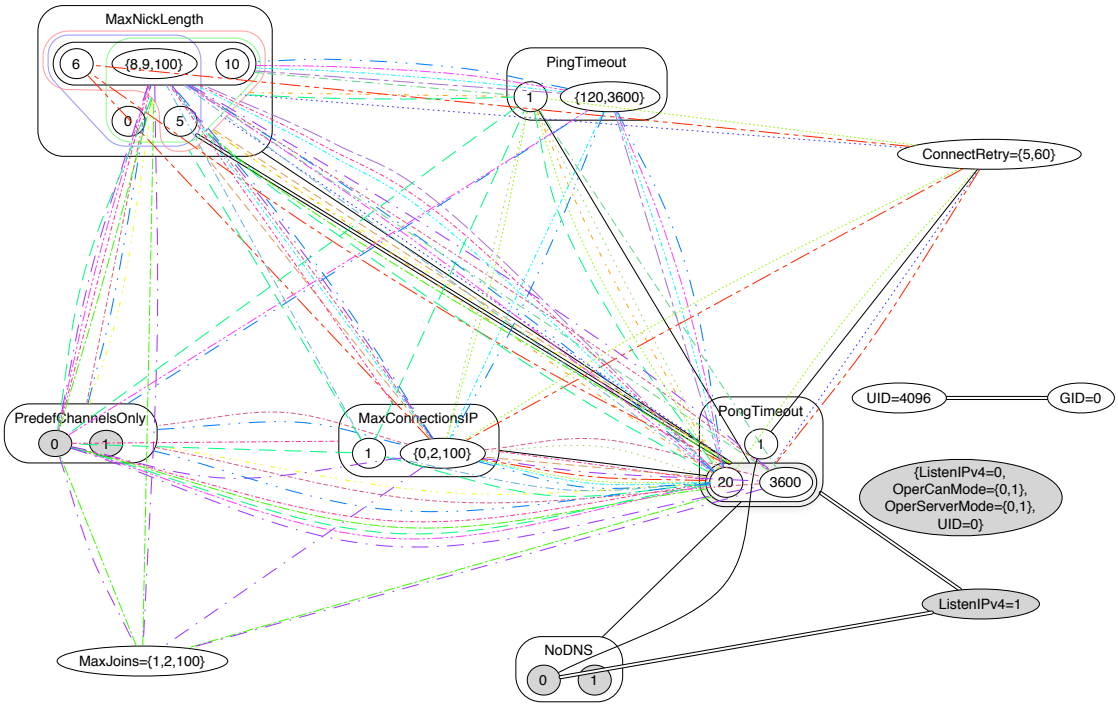


Figure B.1: All line-coverage interactions for ngIRCd. Thin-edge cliques implicitly include ListenIPv4=1.

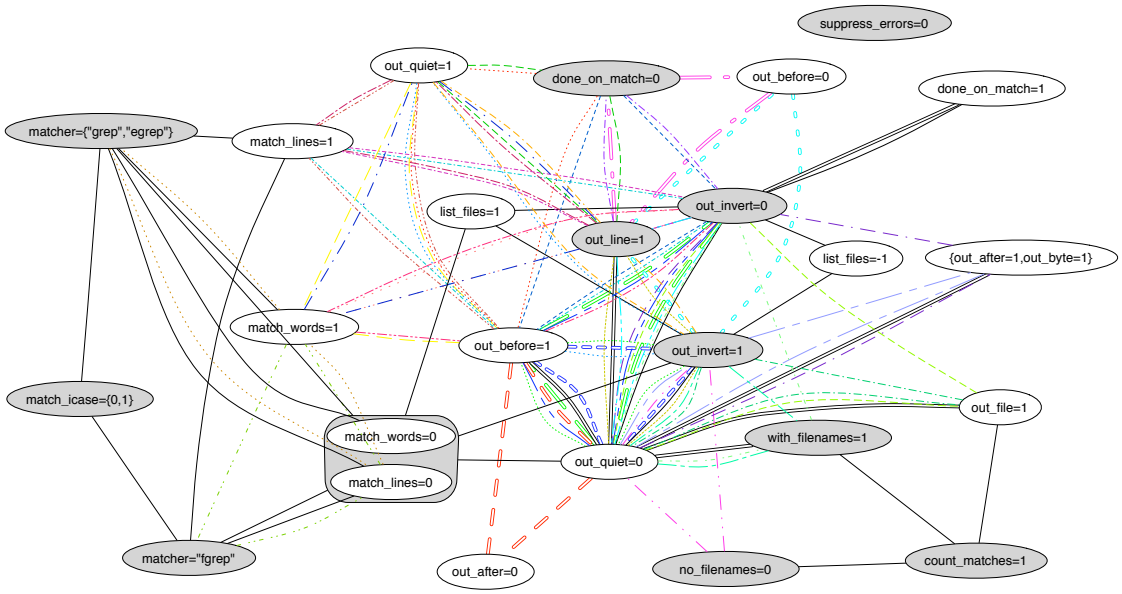


Figure B.2: All line-coverage interactions for grep. Thick-edge cliques implicitly include match_words=0,match_lines=0.

no constraints during symbolic evaluation. For these unconstrained options, we manually selected the values to use in the guaranteed coverage calculations and in Figures B.1 through B.3, as described in section 4.7.1.

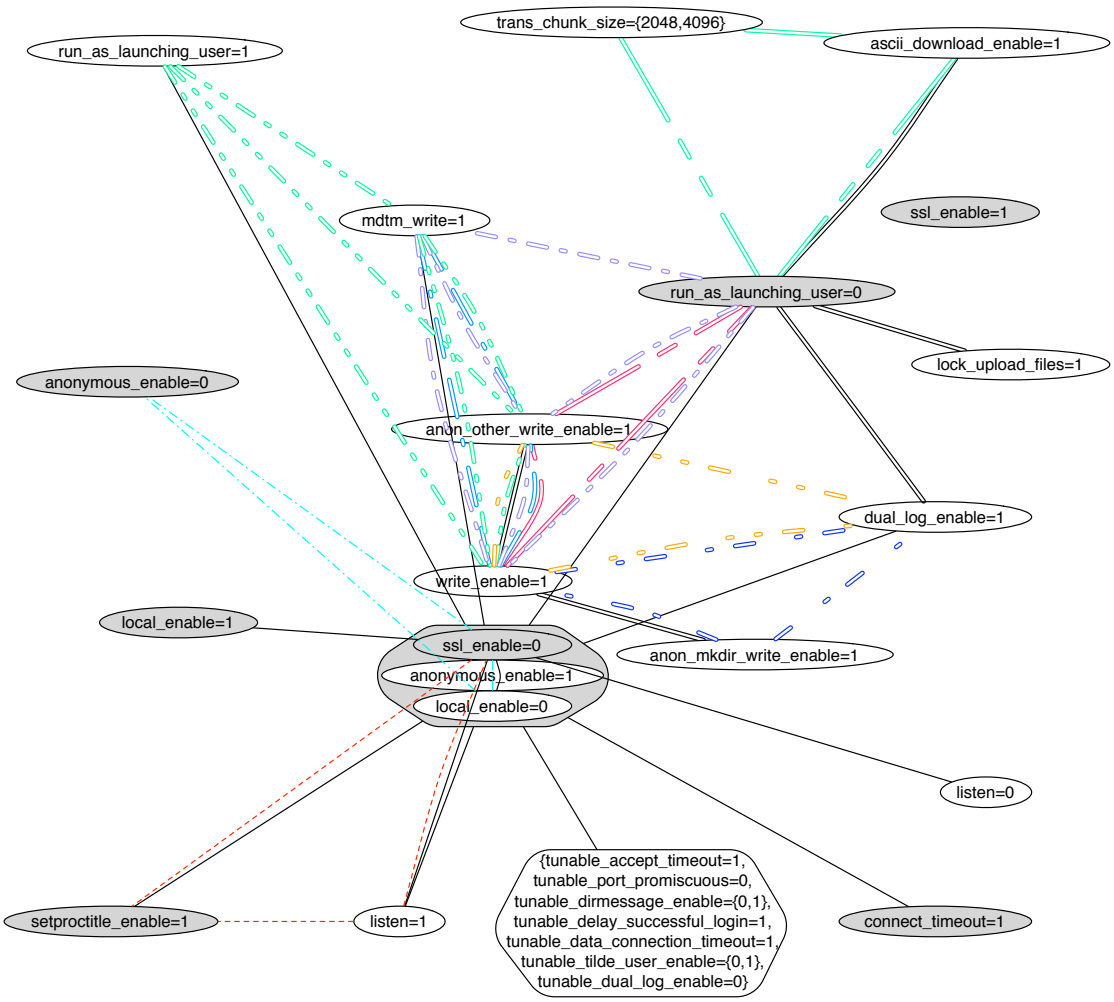


Figure B.3: All line-coverage interactions for vsftpd. Thick-edge cliques implicitly include ssl_enable=0,local_enable=0,anonymous_enable=1.

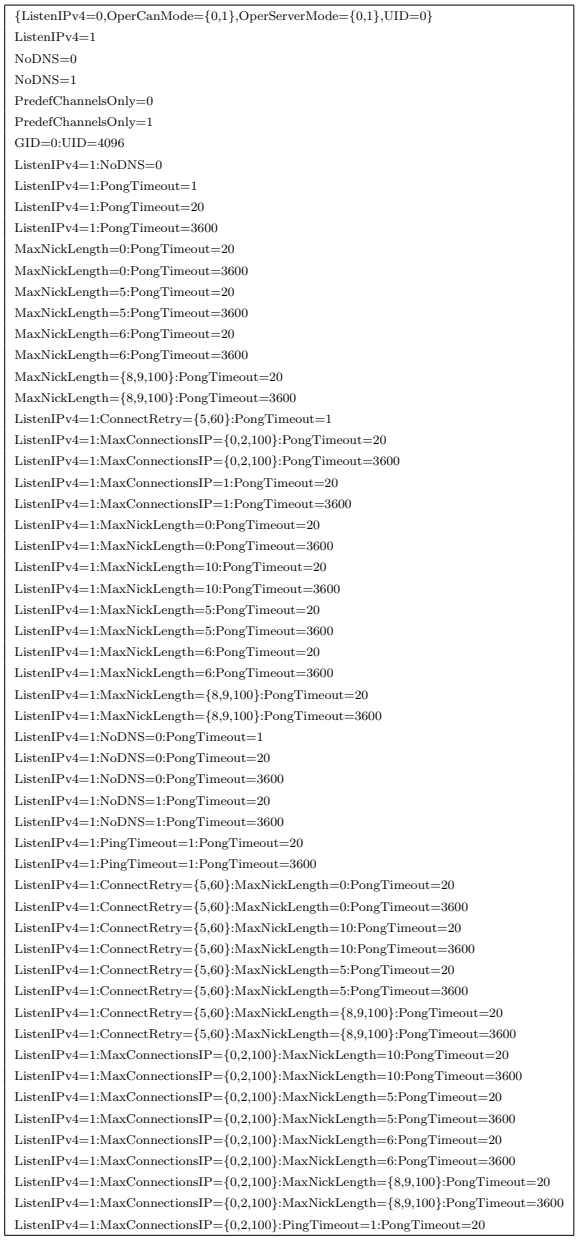


Figure B.4: ngIRCd interactions

```
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=10:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=6:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength={8,9,100}:PongTimeout=20
ListenIPv4=1:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxNickLength=10:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength=10:PingTimeout=1:PongTimeout=3600
ListenIPv4=1:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=5:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength=5:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=5:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=5:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=5:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxNickLength=6:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength=6:PingTimeout=1:PongTimeout=3600
ListenIPv4=1:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout=1:PongTimeout=3600
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:ConnectRetry={5,60}:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=20
ListenIPv4=1:ConnectRetry={5,60}:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=3600
ListenIPv4=1:ConnectRetry={5,60}:MaxConnectionsIP={0,2,100}:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=5:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=5:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxJoins={1,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxJoins={1,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxJoins={1,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=10:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=5:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=6:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength={8,9,100}:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
```

Figure B.5: ngIRCd interactions, continued

Figure B.6: grep interactions

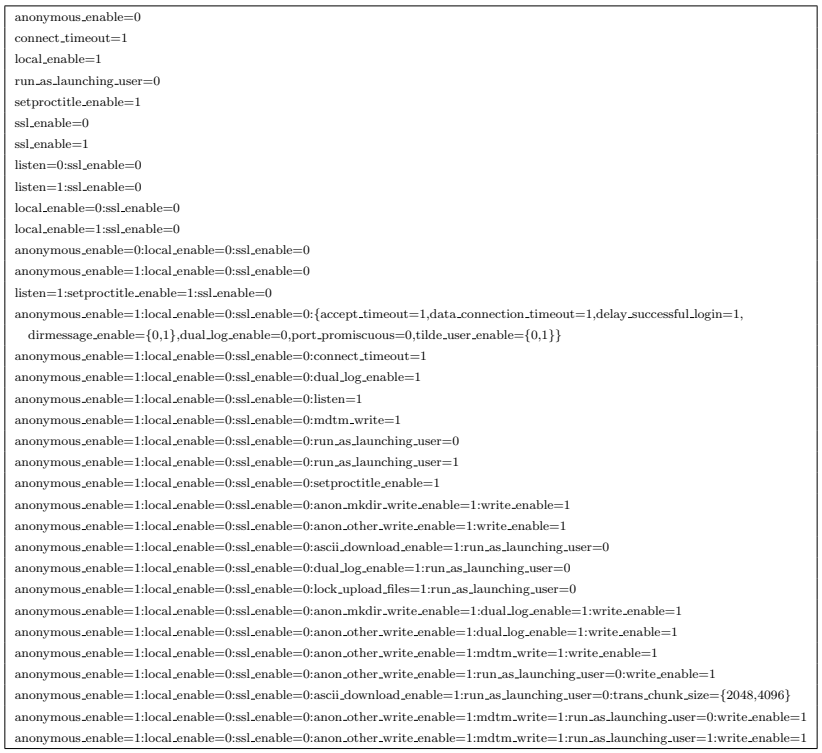


Figure B.7: vsftpd interactions

Name	vsftpd	ngIRCd	grep
Booleans			
	anon_mkdir_write_enable	ListenIPv4	count_matches
	anon_other_write_enable	NoDNS	done_on_match
	anon_upload_enable	OperCanMode	filename_mask*
	anonymous_enable	OperServerMode	match_icase
	ascii_download_enable	PredefChannelsOnly	match_lines
	ascii_upload_enable*		match_words
	delete_failed_uploads*		no_filenames
	dirmessage_enable		out_byte
	dual_log_enable		out_file
	listen		out_invert
	local_enable		out_line
	lock_upload_files		out_quiet
	mdtm_write		suppress_errors
	pasv_addr_resolve*		with_filenames
	port_promiscuous		
	run_as_launching_user		
	setproctitle_enable		
	ssl_enable		
	tilde_user_enable		
	write_enable		
Other			
	accept_timeout	ConnectRetry ∈ {5,60}	list_files ∈ {-1,0,1}
	chown_upload_mode*	GID	matcher ∈ {“grep”, “egrep”, “fgrep”}
	connect_timeout	MaxConnectionsIP	out_after ∈ {0,1}
	data_connection_timeout	MaxJoins	out_before ∈ {0,1}
	delay_successful_login	MaxNickLength	
	ftp_data_port*	PingTimeout ∈ {1,20,3600}	
	listen_port*	PongTimeout ∈ {1,20,3600}	
	max_clients	UID	
	max_per_ip		
	trans_chunk_size		

Figure B.8: Symbolic configuration options. Asterisks indicate options that never led to branching during symbolic evaluation.

Bibliography

[1] Busybox. `http://busybox.net/`.

[2] Gnu c library. `http://www.gnu.org/s/libc/`.

[3] The java pathfinder wiki. `http://babelfish.arc.nasa.gov/trac/jpf`.

[4] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Jpf-se: A symbolic execution extension to java pathfinder. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, pages 134–138, Braga, Portugal, March 2007.

[6] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *POPL*, pages 1–3, 2002.

[7] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229 –245, 1983.

[8] Richard Bornat. Proving pointer programs in Hoare logic. In *International Conference on Mathematics of Program Construction (MPC)*, pages 102–126, 2000.

[9] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[10] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.

[11] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[12] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS*, pages 322–335, 2006.

[13] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *TSE*, 23(7):437–44, 1997.

[14] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE*, pages 38–48, 2003.

[15] Coreutils - GNU core utilities. `http://www.gnu.org/software/coreutils/`.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: Efficient SMT solver. In *TACAS*, volume 4963/2008 of *LNCS*, pages 337–340, 2008.

[17] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallo ws, and A. Iannino. Applying design of experiments to software testing. In *ICSE*, pages 205–215, 1997.

[18] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Software Tools for Technology Transfer*, 5(2):247–267, 2004.

[19] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Trail-directed model checking. *Electrical Notes Theoretical Computer Science*, 55(3):343–356, 2001.

[20] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[21] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.

[22] Patrice Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[23] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[25] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Active property checking. In *EMSOFT*, pages 207–216, 2008.

[26] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[27] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *ISSTA*, pages 12–21, 2002.

[28] James C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[29] The KLEE Symbolic Virtual Machine. `http://klee.llvm.org`.

[30] John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 207–218, 2004.

[31] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[32] Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an AI planning heuristic for directed model checking. In Antti Valmari, editor, *SPIN*, volume 3925 of *LNCS*, pages 35–52. Springer Berlin / Heidelberg, 2006.

[33] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[34] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO*, pages 75–86, 2004.

[35] London Stock Exchange (LSE) system failure stops trading. http://www.zdnet.com/blog/projectfailures/london-stock-exchange-lse-system-failure-stops-trading/472.

[36] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. Technical Report CS-TR-4979, UMD-College Park, Apr 2011.

[37] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[38] R. Mandl. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054–1058, 1985.

[39] Joe M. Morris. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In M Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51, 1982.

[40] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction (CC)*, pages 213–228, 2002.

[41] The Newlib Homepage. `http://sourceware.org/newlib/`.

[42] The Otter Homepage. `http://www.cs.umd.edu/projects/PL/Otter/`.

[43] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.

[44] Adam Porter, Cemal Yilmaz, Atif M. Memon, Douglas C. Schmidt, and Bala Natarajan. Skoll: A process and infrastructure for distributed continuous quality assurance. *TSE*, 33(8):510–525, August, 2007.

[45] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[46] The R Project for Statistical Computing, 2011. `http://www.r-project.org/`.

[47] beeswarm: an R package, 2011. `http://www.cbs.dtu.dk/~eklund/beeswarm/`.

[48] Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–66, 2001.

[49] Elnatan Reisner, Charles Song, Kin-Keun Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, pages 445–454, 2010.

[50] Thomas W. Reps. Program analysis via graph reachability. In *ILPS*, pages 5–19, 1997.

[51] RTI. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3. National Institute of Standards and Technology, Research Triangle Park, NC, May 2002.

[52] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE-13*, pages 263–272, 2005.

[53] Toyota says software glitch in data boxes can give faulty speed readings. http://www.autoweek.com/article/20100914/CARNEWS/100919945#ixzz1JHXYin8Q.

[54] μClibc. `http://www.uclibc.org/`.

[55] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, pages 359–368, 2009.

[56] Cristian Zamfir. Personal communication, May 2011.

[57] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.